Hewlett Packard Enterprise

Enscribe Programmer's Guide

Abstract

This manual describes how to create, access, and load the five types of disk files supported by the Enscribe software.

Acknowledgments

Intel®, Itanium®, Pentium®, Intel Inside®, and the Intel Inside logo are trademarks of Intel Corporation in the United States and other countries.

Microsoft® and Windows® are trademarks of the Microsoft group of companies.

Adobe® and Acrobat® are trademarks of Adobe Systems Incorporated.

Java and Oracle are registered trademarks of Oracle and/or its affiliates.

UNIX® is a registered trademark of The Open Group.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:

© 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991,

1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California. Printed in the US.

Contents

Index...186

What's New in This Manual

Manual Information

Abstract

This manual describes how to create, access, and load the five types of disk files supported by the Enscribe software.

Product Version

Enscribe 1.0

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.27 and all subsequent G-series RVUs until otherwise indicated by its replacement publications. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with specific SPRs. For a list of the required SPRs, see SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release (page 18).

Document History

Part Number	Product Version	Published
520369–009R	Enscribe 1.0	November 2015
520369–009	Enscribe 1.0	June 2014
520369-007	Enscribe G09	July 2012
520369-006	Enscribe G09	February 2012
520369-005	Enscribe G09	February 2011
520369-004	Enscribe G09	February 2011
520369-003	Enscribe G09	February 2006
520369-002	Enscribe G09	December 2002

New and Changed Information

Changes to the 520369-009R Version of the Manual

- Updated Hewlett Packard Enterprise references.

Changes to the 520369-009 Version of the Manual

- Throughout document, change "classic key-sequenced" to "legacy key-sequenced (LKS)".

- Throughout document, update increased partition size, block size, record length, key length, and file size for format 2 legacy key-sequenced files (LKS2), format 2 enhanced key-sequenced files (EKS), and standard format 2 Queue files.

- Update definition of primary and secondary extent sizes for EKS files in Unique Features of EKS Files (page 69) and in the table in Comparing LKS Files, LKS2 Files, Standard Queue Files, EKS Files, and EKS Files with Increased Limits (page 70).

- Add note about support for increased limits in Overview and Features (page 18) and Enscribe Key-Sequenced Files (page 67).

- Add paragraph about increased file size limits in Key-Sequenced and Queue Files (page 47).

- Add table of SPRs required to achieve increased limits with H06.28 and J06.17 in SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release (page 18).

Changes to the 520369-007 Version of the Manual

- Added a note about the Restrictions on Partitioned Unstructured files (page 61).

Changes to the 520369-006 Version of the Manual

- Added new procedures for 64-bit to File-System Procedures (page 38).
- The following chapters have been updated to include the new procedures for 64- bit:

  ○ Introducing the Enscribe Record Manager (page 18)

  ○ Positioning Within Structured Files (page 25)

  ○ System Procedures (page 38)

  ○ General File Creation and Access Information (page 45)

  ○ Unstructured Files (page 59)

  ○ Key-Sequenced Files (page 67)

  ○ Queue Files (page 106)

  ○ Entry-Sequenced Files (page 122)

  ○ Relative Files (page 133)

  ○ File and Record Locking (page 150)

Changes to the 520369-005 Version of the Manual

Added a note to File Locking on page 10-3 about avoiding the frequent use of LOCKFILE and UNLOCKFILE requests on partitioned files due to the significant overhead of these requests and using record level locking instead.

Changes to the 520369-004 Version of the Manual

- These topics have been updated to support Enscribe 64 partitions which is introduced as of H06.22 and J06.11 and later RVUs.

  - Overview and Features (page 18)

  - Partitioned (Multiple-Volume) Files (page 21)

  - File Formats Supported: Format 1 and Format 2 (page 45)

  - Files Secured With Enhanced File Privileges (page 47)

  - Key-Sequenced and Queue Files (page 47)

  - Enscribe Key-Sequenced Files (page 67)

  - Creating Key-Sequenced Files (page 70)

  - Comparing LKS Files, LKS2 Files, Standard Queue Files, EKS Files, and EKS Files with Increased Limits (page 70)

  - Converting a Legacy Key-Sequenced File to an Enhanced Key-Sequenced File (page 72)

  - Key-Sequenced Format 2 Files (page 72)

    - Alternate Keys and Record Locking on page 2-17 and associated topics:

- These topics have been updated in response to Genesis cases:

  - Alternate Keys and Record Locking (page 35) and associated topics:

    - Record Locking Requests and Alternate Key Files (page 35)

    - Implementation of Updates to Alternate Key Records (page 35)

    - Transaction Aborts, Alternate Keys, and Locks (page 36)

    - SETMODE 4 (Set Lock Mode) and Alternate Key Files (page 36)

    - SETMODE 4,6 and SETMODE 4,7 (Read through Lock with Warning) (page 36)

  - Dequeuing a Record (page 110)

  - Dequeuing From Audited Files (page 111)

  - Approximate Positioning (page 30)

  - Generic Positioning (page 30)

  - Corrected FILE_SETKEY to FILE_SETKEY_ in several sections of the manual.

  - Corrected minor grammatical and typographical errors reported via Genesis.

- Added hyperlinked topics to the beginning of every chapter to make it easier to locate information within a chapter.

Changes to the 520369-003 Version of the Manual

- Clarified the behavior described under Reading Deleted Records (page 158), for the alternate and primary key access.

- Added a note on End-of-File values in Table 6 and added footnote.

- Rebranded the manual with the latest terminology.

Changes to the G06.18 Manual

- Added a caution note in Section 7 (Queue Files) to advise the user that the use of queue files with multiple sub-queues can cause high utilization of CPU resources by the disk process and affect performance.

- Added a Performance Considerations section to discuss the practical limits on how many processes should be used when multiple servers (dequeuing processes) are used on a Queue File.

- Changed a table entry to read "4 GB – 4 KB". Also, changed the table entry for maximum unpartitioned file size to "2 GB – 1 MB".

About This Manual

This manual documents the Enscribe database record manager. It is written for programmers, database designers, and analysts whose job is to design, develop, and maintain database applications to be executed on HPE NonStop systems.

Organization of This Manual

Related Manuals

These NonStop manuals provide additional information that you might want to have available for reference:

- *Guardian Procedure Calls Reference Manual*

- *Guardian Programmer's Guide*

- *Guardian Disk and Tape Utilities Reference Manual*

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. Computer type letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type.

```
Italic computer type
```

letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example

```
TERM [\system-name.]$terminal-name
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LIGHTS [ ON ]
[ OFF ]
[ SMOOTH [ num ] ]
K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name }
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address-1 [ , new-value ]...
[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate that the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id );
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] CONTROLLER

[ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT ( segment-id                !I
                        , error            ) ;            !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;            !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ ( filename1:length        !i:i
                           , filename2:length ) ;    !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ ( filenum                    !i
                       , [ filename:maxlen ] ) ;    !o:i
```

Notation for Messages

The following list summarizes the notation conventions for the presentation of displayed messages in this manual

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED:            123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

```
p-register
```

```
process-name
```

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number = number [ Subject = first-subject-value

]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list might be arranged either vertically,

with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
 LDEV ldev [ CU %ccu | CU %... ] UP [ (cpu,chan,%ctlr,%unit) ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list might be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }

process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown. }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The %þnotation precedes an octal number. The %Bþnotation precedes a binary number. The %Hþnotation precedes a hexadecimal number. For example:

```
%005400

%B101111

%H2F

P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

The following list summarizes the notation conventions used in the boxed descriptions of error lists in this manual

UPPERCASE LETTERS. Uppercase letters indicate names from definition files; enter these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r.. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME           token-type ZSPI-TYP-STRING.             !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER                 token-type ZSPI-TYP-FNAME32.        !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, s, tables, examples, and so on. Change bars highlight new or revised information. For example:

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Introducing the Enscribe Record Manager

Overview and Features

SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release

As of the H06.28 and J06.17 RVUs, format 2 legacy key-sequenced 2 (LKS2) files with increased limits, format 2 standard queue files with increased limits, and enhanced key-sequenced (EKS) files with increased limits are introduced. EKS files with increased limits support 17 to 128 partitions along with larger record, block, and key sizes. LKS2 files with increased limits and format 2 standard queue files with increased limits support larger record, block, and key sizes. When a distinction is not required between these file types, key-sequenced files with increased limits is used as a collective term. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with the following specific SPRs. (These SPR requirements could change or be eliminated with subsequent RVUs.)

Table 1 SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release

Products	J-Series SPR	H-Series SPR
Backup/Restore NSK	T9074H01^AGJ	T9074H01^AGJ
DP2	T9053J02^AZZ	T9053H02^AZN
File System	T9055J05^AJQ	T9055H14^AJP
FUP	T6553H02^ADH	T6553H02^ADH
NS TM/MP TMF DR	T8695J01^ALP	T8695H01^ALO
SMF	T8472H01^ADO T8471H01^ADO T8470H01^ADO T8469H01^ADO T8466H01^ADO T8465H01^ADO T8468H01^ABY	T8472H01^ADO T8471H01^ADO T8470H01^ADO T8469H01^ADO T8466H01^ADO T8465H01^ADO T8468H01^ABY
SQL/MP	T9191J01^ACY T9195J01^AES T9197J01^AEA	T9191H01^ACX T9195H01^AER T9197H01^ADZ
TCP/IP FTP	T9552H02^AET	T9552H02^AET
TNS/E COBOL Runtime Library	T0357H01^AAO	T0357H01^AAO

NOTE: As of the H06.22 and J06.11 RVUs, a new Enscribe file type: enhanced key-sequenced (EKS) file is introduced and extends the Enscribe partition maximum to 64. An EKS file supports 17 to 64 partitions. The previous Enscribe key-sequenced file type supports 1 to 16 partitions and is referred to as a legacy key-sequenced (LKS) file in this manual. When a distinction is not required between these file types, key-sequenced file is used as a collective term.

The Enscribe record manager, supported by the Guardian file system, provides high-level access to records in a database. Distributed across two or more processors, the software helps ensure data integrity if a processor module, I/O channel, or disk drive fails.

This manual describes the use of the software with the DP2 disk process. Some of the most notable Enscribe features are:

- Five disk file structure types:
  - unstructured
  - key-sequenced:
    - legacy key-sequenced (1 to 16 partitions, file format 1 and 2)
    - legacy key-sequenced with increased limits (1 to 16 partitions, larger record size, larger block size, larger key size, file format 2 only)
    - enhanced key-sequenced (17 to 64 partitions, file format 2 only)
    - enhanced key-sequenced with increased limits (17 to 128 partitions, larger record size, larger block size, larger key size, file format 2 only)
  - queue
  - entry-sequenced
  - relative
- File Formats: format 1 and format 2
- Partitioned (multiple-volume) files
- Multiple-key access to records
- Relational access among files (where a field value from one file is used as a key to access a data record in another file)
- Optional automatic maintenance of all keys and optional key compression in key-sequenced data or index blocks
- Support of transaction auditing through the NonStop Transaction Management Facility (TMF).
- Optional compression of audit-checkpoint records
- Record locking and file locking
- Cache buffering
- Optional sequential block buffering

Terminology

To create and use Enscribe files, you should be familiar with these terminologies:

- File: a collection of related records, physically organized as a number of extents on disk, that is referenced by a Guardian file name. Structured files (entry-sequenced, key-sequenced, queue, and relative files) are further subdivided into blocks of records.
- Format 1 and 2: Format is a static attribute of a file and is established when the files is created. Format 2 files, introduced with the D46 release, differ from format 1 files in these ways: larger partitions than the current 2 GB minus 1 MB format 1 file partition and larger primary keys and alternate-key records for relative and entry-sequenced files.
- Logical record: the unit of information transfer between your application program and the file system; unless you are using sequential block buffering, the logical record is also the unit of information transfer between the file system and the disk process. The maximum logical record size is specified as a number of bytes.
- Sector: the smallest unit of disk I/O or physical addressing (512 bytes in length for all currently supported disk drives).

- Extents: a unit of storage allocation for a file to be allocated as contiguous space on disk. Each extent consists of some specified number of 2048-byte pages.

- Block: the unit of physical I/O. A block consists of one or more logical records plus some control information for structured files. When you are using sequential block buffering, a block is the unit of transfer between the file system and the disk process. The block size is specified as a number of bytes. The block sizes currently supported are 512 bytes, 1 KB, 2 KB, and 4 KB. 32 KB block size is only supported for key-sequenced files with increased limits.

- Key: a value associated with a record (a record number, for example) or contained within a record (as a field) that can be used to locate one record or a subset of records in a file.

Disk File Organization

A disk file must be created before it can be accessed. You can create a file either by calling the FILE_CREATE_ procedure or by using the File Utility Program (FUP) CREATE command, then designate the file as either permanent or temporary. A permanent file remains in the system after access is terminated; a temporary file is deleted when all its openers have closed it. You also specify the file's type when you create it. Taken as a group, key-sequenced, queue, entry-sequenced, and relative files are known as structured files. The facilities available with structured files differ significantly from those available with unstructured files.

The disk process allocates physical storage to files in the form of extents, each consisting of some number of contiguous 2048-byte pages on the disk.

A partitioned file (one having extents on more than one disk volume) other than a key-sequenced file is limited to 16 extents per partition. The maximum number of extents in a non partitioned file or a partitioned file is restricted by the maximum label size up to a limit of 978 extents per file (or per partition for partitioned key-sequenced files). The maximum label size is further constrained by the number of alternate keys and partitions defined for this file, by the file format, and by whether the file is an SMF file. Within this limit, you can use the MAXEXTENTS attribute to set an arbitrary limit for any nonpartitioned file (or for any partition of a key-sequenced file).

NOTE: For all Enscribe file types, the disk process constrains the use of MAXEXTENTS such that any partition is always smaller than two gigabytes if the file is format 1.

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 disk space as the file grows.

An application process can allocate one or more extents in an open file by way of a CONTROL 21 procedure call. The CONTROL procedure can also deallocate unused extents.

Unstructured Files

An unstructured disk file is essentially a large byte array. Most often it is used as a code file or an edit file, not as a data file. The organization of an unstructured disk file (the lengths and locations of records within the file) is entirely 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.

Structured Files

The Enscribe product provides four types of structured files: key-sequenced, queue, entry-sequenced, and relative. Data transfers between an application process and a structured disk file are done in terms of logical records and record keys.

For key-sequenced files, the primary key is a particular data field, designated by the user, within each data record. For queue files, the primary key resides within the data record and consists of an 8-byte timestamp plus an optional user key. For entry-sequenced files, the primary key is a block number and record number, external to the data record, specifying the record's storage location within the file. For relative files, the primary key is an ordinal number, again external to the data record, denoting the record's position within the file.

In addition to access by primary key, you can specify alternate keys for key-sequenced, entry-sequenced, and relative files. Queue files cannot have alternate keys.

Several NonStop software products, such as Enform and Enable, are available to help you define and access Enscribe structured files.

Partitioned (Multiple-Volume) Files

When you create a file, you can designate it to reside entirely on a single volume or you can have it partitioned over several volumes. Moreover, the separate volumes need not reside on the same system; you can partition a file across network nodes.

Enhanced key-sequenced files can have 17 to 64 parts. For enhanced key-sequenced files with increased limits, 17 to 128 partitions are permitted. Up to 16 partitions are permitted for all other Enscribe file types. Queue files cannot be partitioned. The primary partition of an enhanced key-sequenced file cannot contain user data and is instead used to store a portion of the file's label.

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 the FILE_OPEN_ procedure. Unless you specify unstructured access, the file system rejects any attempt to open a secondary partition of a file.

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 capacity of a physical disk drive. In addition, by spreading a file across more than one volume, you can increase the concurrency of access to records in separate partitions of the file.

If the file is located on multiple volumes on the same controller, the operating system takes 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. If the file resides on volumes connected to controllers on different processors, the system performs 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.

Partitioned file records can also reside in multiple caches, which can result in fewer disk 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 different volume names.

Few Differences Among Partitions

All the partitions of a file must be either audited by TMF or not audited; they cannot be mixed.

Primary and secondary extent sizes can differ from one partition to another within the same partitioned file. In addition, the MAXEXTENTS value can differ from one partition to another within a key-sequenced file. These and the volume names are the only file attributes on which partitions can differ.

File Directory

A disk 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 Subsystem Control Facility (SCF) to label the disk, by estimating how many files you want each directory extent to hold. The system translates this to an approximate extent size when it creates the actual directory file. The actual number of files that 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 might not always be precisely what you specified.

The disk process can potentially create as many as 987 directory extents, so the creation of too many files to fit in the currently allocated directory extent space merely causes the disk process to allocate another directory extent.

△ **CAUTION:** Never write to a file directory. Any attempt by an application process to alter the content of a file directory can cause directory corruption and/or a DP2 halt.

Audited Files

In a system with TMF, any database file can be designated as an audited file. To help maintain database consistency, TMF audits all transactions involving files designated as audited files. That is, TMF maintains images (in an audit trail) of the database changes made by those transactions. If necessary, TMF can use the audit trail later to back out failed transactions or to restore audited files that some system failure has rendered inconsistent.

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

Access Coordination

Several different processes can have access to one file at the same time. For coordination of simultaneous access, each process must indicate, when opening a file, how it intends to use that file. Each process must specify both an access mode and an exclusion mode.

The access mode specifies which operations are to be performed. The access mode can specify read/write (the default access mode), read-only, or write-only access.

The exclusion mode specifies how much access is granted to other processes. The exclusion mode can specify shared, exclusive, or protected access.

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

Waited and Nowait I/O

The software allows an application process to execute concurrently with its file operations by means of nowait I/O.

The default is waited I/O; when designated file operations are performed, the application process is suspended until the operation completes.

Nowait I/O means that when designated file operations are performed, the application process is not suspended. 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.

Waited and nowait I/O are described in the *Guardian Programmer's Guide*. For more information, also see the descriptions of the FILE_OPEN_, READ, FILE_READ64_, AWAITIO and FILE_AWAITIO64_ procedures in the *Guardian Procedure Calls Reference Manual*.

Operations on Files

Common file operations include creating files, describing record formats, loading files, and manipulating records.Table 2 (page 23) summarizes the record management functions that are most commonly used with Enscribe files.

Creating Files

You create disk files either by calling the FILE_CREATE_ system procedure or by issuing FUP commands:

- Programmatic creation of disk files is accomplished by supplying the appropriate parameters to the FILE_CREATE_ or FILE_CREATELIST_ procedure.

- The FUP commands SET, RESET, and SHOW let you specify, display, and modify creation parameters (such as file type, record length, key description, and so forth) before actually creating a file. If you like, you can set the creation parameters to be like those of an existing file. The FUP CREATE command then creates a file with the currently set parameters. The ALTER command lets you change some of those parameters after the file is created. FUP accepts commands from an online terminal or from a command (OBEY) or IN file.

Loading Files

You can use FUP to load data into existing Enscribe files. You specify the set of records to be loaded and the file's data block 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.

Manipulating Records

You can manipulate records in an Enscribe file by calling file-system procedures. Record management functions and the associated system procedures are summarized in Table 2 (page 23)

Table 2 Record Management Functions Summary

Function	Description	Procedures
Delete	Deletes the record in a key-sequenced, queue, or relative file as indicated by a primary-key value.	FILE_WRITEUPDATE64_, WRITEUPDATE
Find	Sets 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.	FILE_SETPOSITION_, FILE_SETKEY_, KEYPOSITION, POSITION
Insert	Inserts a new record into a file according to its primary-key value.	FILE_WRITE64_, WRITE
Lock	Locks the whole file, just the current record, or a set of records containing the same generic key.	FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_READLOCK64_, FILE_READUPDATELOCK64_, LOCKREC, LOCKFILE, READLOCK, READUPDATELOCK
Read	Reads a subset of records sequentially.	FILE_READ64_, READ
Unlock	Unlocks the whole file, the current record in a file, or all the records in a file.	FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, FILE_WRITEUPDATEUNLOCK64_,

Table 2 Record Management Functions Summary *(continued)*

Function	Description	Procedures
		UNLOCKREC, UNLOCKFILE, WRITEUPDATEUNLOCK
Update	Updates a record in a random position in a file.	FILE_READUPDATE64_, FILE_WRITEUPDATE64_, READUPDATE, WRITEUPDATE

Comparison of Structured File Characteristics

The Enscribe product provides four types of structured files: key-sequenced files, queue files, entry-sequenced files, and relative files. compares the characteristics of the four types of structured files.

Table 3 Comparison of Structured Files

Key-Sequenced	Queue	Entry-Sequenced	Relative
Records are ordered by value in primary-key field	Records are ordered first by user key (if present) and then by timestamp	Records are in the order in which they are entered	Records are ordered by relative record number
Access is by primary or alternate key	Access is by primary key (consisting of an optional user key and a timestamp).	Access is by record address or alternate key	Access is by record number or alternate key
Length of primary key varies. Key is actually part of record.	Length of primary key varies but must be at least 8 bytes to hold a timestamp.	Record address is primary key. Length: 8 bytes.	Record number is primary key. Length: 8 bytes.
The software uses index blocks to locate primary key, which is stored in the record.	The software uses index blocks to locate primary key, which is stored in the record.	The software uses record address to find physical location of record in file.	The software uses record number to calculate the physical location of record in file.
Space occupied by a record depends on length specified when written.	Space occupied by a record depends on length specified when written.	Space occupied by a record depends on length specified when written.	Space allowed per record is specified when the file is created.
Free space in block or at end of file is used for adding records.	Free space in block or at end of file is used for adding records.	Space at end of file is used for adding records.	Empty positions in file are used for adding records.
Records can be deleted, shortened, or lengthened (within the maximum size specified).	Records can only be deleted.	Records cannot be deleted, shortened, or lengthened.	Records can be deleted, shortened, or lengthened (within the maximum size specified).
Space freed by deleting or shortening a record can be reused.	Space freed by deleting a record can be reused.	deleted, but its space can be used for another record of the same size.	Space freed by deleting a record can be reused.

2 Positioning Within Structured Files

Structured File Records

A record in an Enscribe structured file consists of one or more data fields, as illustrated in Figure 1 (page 25)

Figure 1 A Record With Three Fields in a Structured File

CUSTOMER Record

Name	Address	Region

Fields
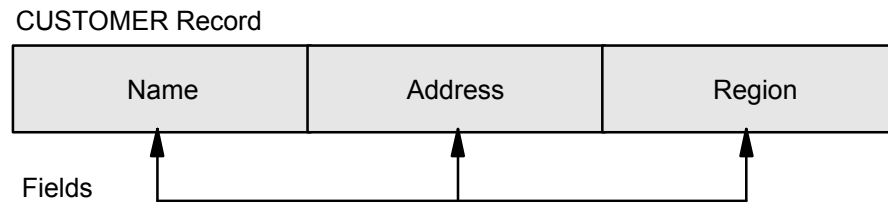
Each record in a structured file is uniquely identified among other records in that file by the value of its primary key. Figure 2 (page 26) illustrates the primary keys for the four different types of structured files:

- The primary key for key-sequenced files is a particular data field within each record.

- The primary key for a queue file consists of a user key and a timestamp within each record.

- The primary key for entry-sequenced files is a record address maintained by the software.

- The primary key for relative files is a record number maintained by the software.

The records within structured files are stored in ascending order by primary-key value.

Figure 2 Primary Keys in Structured Files

Key-Sequenced Files:

Name	Address	Region

Primary Key

Queue Files:

Mfg. Site	Timestamp	Description	Part Number

Primary Key

Entry-Sequenced Files:

<record address>

Name	Account Number

Primary Key

Relative Files:

<record number>

Item	Description

Primary Key
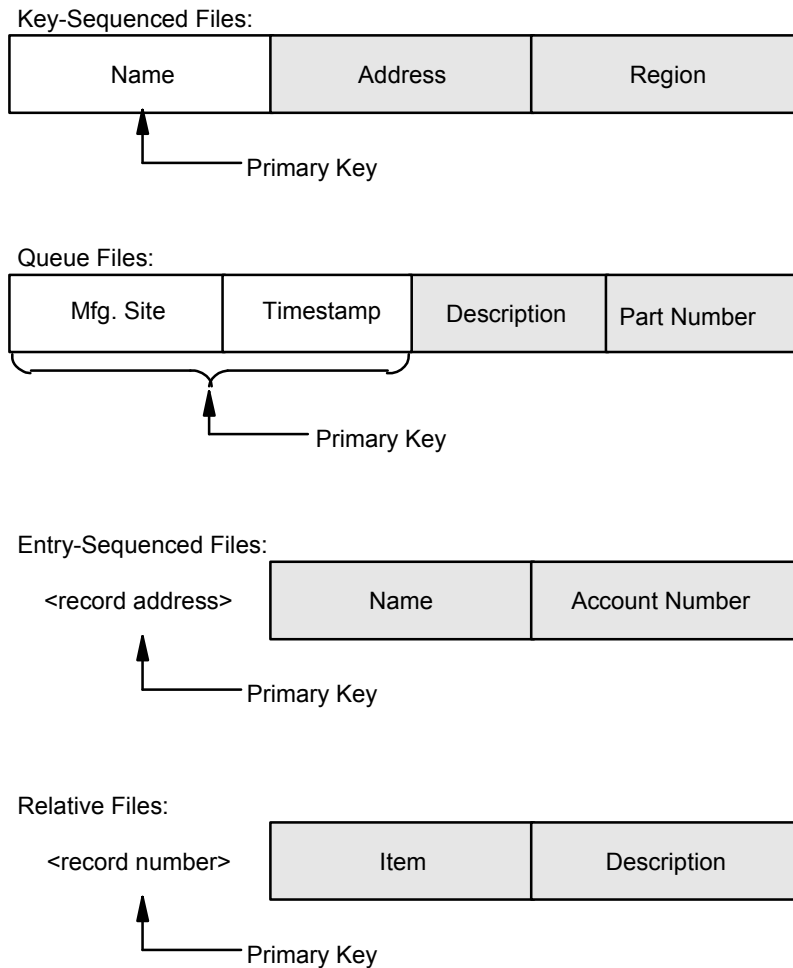
Except for queue files, structured files can also include up to 255 alternate-key fields. As illustrated in Figure 3 (page 26), an alternate key is a designated data field within the record whose values can be used at execution time to logically subdivide the overall file into meaningful subsets of records. In an employee data file, for example, one data field in the record format might be the employee's region code; if you define that field as an alternate key, you can access only those data records that pertain to the employees who all reside in a particular geographic region.

As illustrated in Figure 4 (page 27), primary-key values are always unique, but alternate-key values can be the same from one record to another.

Figure 3 An Alternate-Key Field

Name	Address	Region

Alternate-Key Field
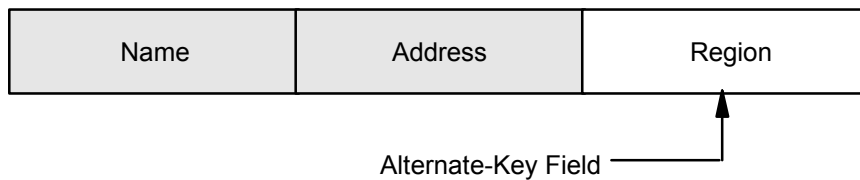
The software lets you use the primary-key value to locate one unique record among other records in the same file. For example, in Figure 4 (page 27), the name field is the primary key and the primary-key value JONES, J.A. locates the only record having that name.

By using alternate-key values, you can process a subset of records that all contain the same value in a particular data field. For example, using Figure 4 (page 27) again, the REGION field is defined as an alternate key and the value CENTRAL provides you with access to two records.

Figure 4 Using Key Values to Locate Records

In a key-sequenced file, only one record can have the primary-key value JONES, J.A.

JONES, J.A.	DAYTON, OHIO	CENTRAL
MOORE, Q.A.	LOS ANGELES, CALIFORNIA	WESTERN
SMITH, S.A.	CHICAGO, ILLINOIS	CENTRAL

Two or more records, however, can have the value CENTRAL in their region alternate-key fields
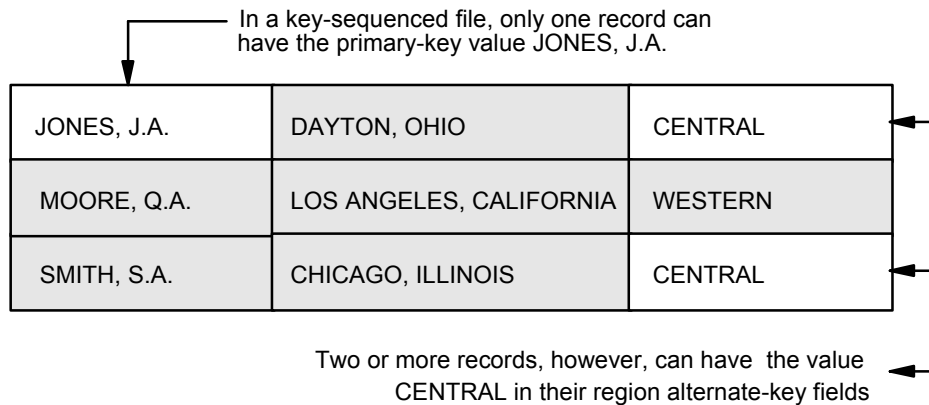
Access Paths

Each key in a structured file provides a separate access path through the records in that file. Records in any given path are accessed by ascending key values. In the case of duplicate alternate-key values, the records containing the same key value are accessed within that path in the order in which they appear in the alternate-key file.

Normally, duplicate alternate-key records are stored in the alternate-key file in ascending order by their associated primary-key values. When you create an alternate-key file, however, you can specify (by way of the *alternate-key-params* array) that duplicate alternate-key records be stored in the order in which they are added to the file.

Note that all alternate-key files for any given primary database file must use the same ordering convention for records with duplicate keys. That is, if one alternate-key file for a particular primary file contains insertion-ordered duplicate key records, then all the alternate-key files for that primary file must do so.

When you perform a read or write operation that uses or creates an insertion-ordered duplicate alternate-key record, the file system returns an advisory error code (CCL with a code of 551) upon completion of the particular system procedure call.

Current Key Specifier and Current Access Path

A 2-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 *nullnull*). Key specifiers for alternate-key fields are defined by the application and are assigned when the file is created. Figure 6 (page 29) shows a typical record structure with a primary key and three alternate keys.

The current key specifier defines the current access path, which affects the order of the records of the file (see Figure 5 (page 28)). The current access path is implicitly set to the file's primary key when a file is opened; for entry-sequenced and relative files, the current access path is also set to the primary key of the file when a call is made to the POSITION procedure. The access path is set explicitly by calling the KEYPOSITION procedure.

Figure 5 Access Paths

	Employee Number	Name	Address (Not an Access Path)	Dept
Records in Order by EMPLOYEE NUMBER Access Path	001	RYAN		C
	002	KING		A
	005	FISH		D
	008	ADAMS		B
	010	BROWN		A
	011	STEVENS		C
	012	OBRIEN		D
	013	MASTERS		C
	016	WATSON		B

↑
Access Path

	Employee Number	Name	Address (Not an Access Path)	Dept
Records in Order by NAME Access Path	008	ADAMS		B
	010	BROWN		A
	005	FISH		D
	002	KING		A
	013	MASTERS		C
	012	OBRIEN		D
	001	RYAN		C
	011	STEVENS		C
	016	WATSON		B

↑
Access Path

	Employee Number	Name	Address (Not an Access Path)	Dept
Records in Order by DEPT Access Path	002	KING		A
	010	BROWN		A
	008	ADAMS		B
	016	WATSON		B
	001	RYAN		C
	011	STEVENS		C
	013	MASTERS		C
	005	FISH		D
	012	OBRIEN		D

↑
Access Path

Figure 6 Key Fields and Key Specifiers
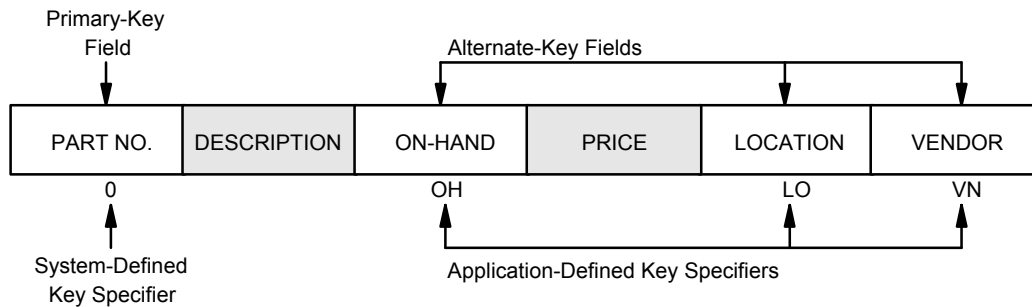
Current Key Value and Current Position

The current key value defines a file's current position. You can set the current key value explicitly by calling the FILE_SETKEY_, FILE_SETPOSITION_,POSITION or KEYPOSITION procedure. FILE_SETKEY_ and KEYPOSITION set a position by primary key for key-sequenced and queue files and by alternate key for key-sequenced, entry-sequenced, and relative files. FILE_SETPOSITION_ and POSITION set a position by primary key for entry-sequenced and relative files. After a call to FILE_READ64_/READ, the current key value is implicitly set to the key value of the current access path in the record just read. Figure 7 (page 29) demonstrates the use of KEYPOSITION in a key-sequenced file.

The current position determines the record to be locked (by a call to FILE_LOCKREC64_/LOCKREC) or accessed (by a call to FILE_READ[LOCK]64_, READ[LOCK], FILE_READUPDATE[LOCK]64_, READUPDATE[LOCK], FILE_WRITEUPDATE[UNLOCK]64_, or WRITEUPDATE[UNLOCK]). A record need not exist at the current position. When a file is opened, the current position is that of the first record in the file as defined by the file's primary key.

Figure 7 Current Position
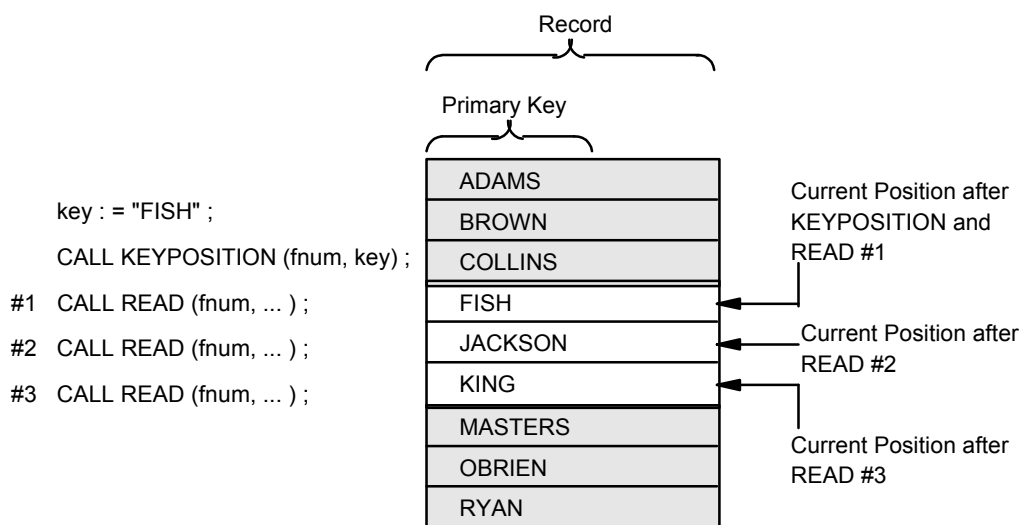
Positioning Mode and Comparison Length

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. Approximate mode selects all records whose access path key values are equal to or greater than the supplied key value.

Generic mode selects all records whose access path key value matches a supplied partial value. Exact mode selects only those records whose access path key value matches the supplied key value exactly. Figure 8 (page 30) shows examples of subsets returned by each positioning mode.

The positioning mode, comparison length, and current key value supplied in a KEYPOSITION procedure call together specify 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, or none of the records in a file.

Figure 8 Approximate, Generic, and Exact Subsets

Approximate	Generic	Exact
Find "JONES"	Find "JONES"	Find "JONES, K.A."

HUNT, H.L.	HUNT, H.L.	HUNT, H.L.
IRWIN, F.J.	IRWIN, F.J.	IRWIN, F.J.
JONES, A.B. (Start)	JONES, A.B. (Start)	JONES, A.B.
JONES, K.A.	JONES, K.A.	JONES, K.A. (Start, Exact Subset)
JONES, M.P.	JONES, M.P.	JONES, M.P.
JONES, Z.Z.	JONES, Z.Z.	JONES, Z.Z.
JORDAN, W.R.	JORDAN, W.R. (Generic Subset)	JORDAN, W.R.
KANE, C.T.	KANE, C.T.	KANE, C.T.
KING, M.L. (Last Record in File)	KING, M.L.	KING, M.L.
LACEY, G.T. (Approximate Subset)	LACEY, G.T.	LACEY, G.T.
LANG, M.N.	LANG, M.N.	LANG, M.N.
MEYER, D.J.	MEYER, D.J.	MEYER, D.J.

| Reading the file sequentially returns 10 records, then an EOF. | Reading the file sequentially returns 4 records, then an EOF. | Reading the file sequentially returns 1 record, then an EOF. |

The positioning mode and comparison length (as well as the current key specifier and current key value) are set explicitly by the FILE_SETKEY_ and KEYPOSITION procedures and implicitly by the FILE_OPEN_, FILE_SETPOSITION_ and POSITION procedures. The software supports three positioning modes:

• approximate

• generic

• exact

Approximate Positioning

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 for the number of bytes specified by the comparison length. After approximate positioning, sequential FILE_READ64_/READ operations to the file return ascending records in the current access path until the last record in the file is read; an EOF indication is then returned. When a file is opened, the positioning mode is set to approximate and the comparison length is set to 0.

Generic Positioning

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 FILE_READ64_/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 EOF indication is returned.

> **NOTE:** For entry-sequenced and relative files, generic positioning by the primary key is the equivalent of exact positioning.

Exact Positioning

Exact positioning means the only records accessed are those whose key field, as designated by the current key specifier, contains a value that is both:

- exactly as long as the specified comparison length

- equal to the current key value.

When the current key no longer matches, an EOF indication is returned. 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

Thus, the length of an alternate-key record in bytes, is:

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

Figure 9 (page 32) shows how alternate keys are implemented.

Figure 9 Alternate-Key Implementation

Primary Data file Record Format:

PART NO.	DESCRIPTION	ON-HAND	PRICE	LOCATION	VENDOR

Primary-Key → PART NO.

Alternate-Key → ON-HAND, LOCATION, VENDOR

0 (Key Specifier) under PART NO.
OH under ON-HAND
LO under LOCATION
VN under VENDOR

Alternate-Key File Record Format (Access Path OH):

OH	ON-HAND	PART NO.

Key Specifier → OH

Field Value → ON-HAND

Primary-Key Value → PART NO.

Sample Data in Primary Data File:

PART NO.	DESCRIPTION	ON-HAND	PRICE	LOCATION	VENDOR
0115	TOASTER	20	12.50	C	TWR
0201	T.V. SET	5	200.00	A	ACME
0205	PHONOGRAPH	52	55.00	B	ACR
0206	RADIO	97	9.95	A	BROWN
0310	FRY PAN	19	37.50	D	SMITH
0322	MIXER	12	69.95	D	ACME

Key Specifier

To identify a particular data field as an alternate-key access path, you must differentiate it from other fields by assigning it a 2-byte key specifier (such as OH for the on-hand field). You supply the 2-byte key specifier when you create the primary file.

Key Offset

For each alternate key you must specify its offset from the start of the record (where the alternate-key field begins) when creating or altering characteristics of the primary file.

Consider these when choosing the offset of an alternate-key field:

- An alternate-key field can begin at any offset in the record.

- Alternate-key fields can overlap.

- Alternate-key fields are fixed-length but need not contain a data value (that is, they can contain a null value) 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 specified offset must be such that the field begins on a word boundary.

Automatic Maintenance of All Keys

When a new record is added to a file or a value in an alternate-key field is changed, the software automatically updates the indexes 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.

Null Value

You can assign a null value to any alternate key. You can choose any character as the null value; the most commonly used are ASCII blank (%40) and binary zero.

If a record is inserted and one of its alternate-key fields contains only bytes of the null value, an alternate-key reference is not added to the alternate-key file. If a record is updated and one of its alternate-key fields is changed to contain only bytes of the null value, the associated alternate-key reference is deleted from the alternate-key file.

If a structured file is read sequentially by alternate key, any records containing the null value in that field are skipped. Instead, the next record returned (if any) is the next one not having the null value in that alternate-key field. 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.

Unique Alternate Key

An alternate-key field can be specified to require a unique value in each record. If you try to insert a record that duplicates an existing record's value in a unique key field, the insertion is rejected with an error 10 (`"record already exists"`). When using nonunique alternate keys, such an insertion would be permitted. If a file has one or more unique alternate keys, remember that:

- For each alternate-key field having a unique key-length, you 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; this is illustrated by the sample alternate-key file following Figure 9 (page 32)

No Automatic Update

You 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. Two reasons for doing so are:

- Certain fields might 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. If that is the case, your application program must have the alternate-key file open separately and must determine itself whether or not the field contains a null value. If the field does not contain a null value, your program then inserts the appropriate alternate-key reference into the alternate-key file.

Alternate Keys in a Key-Sequenced File

You might use alternate keys in a key-sequenced 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 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 returns 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 an Entry-Sequenced File

You might use alternate keys in an entry-sequenced file within a transaction-logging file. The primary key (a record address) would indicate the order in which transactions occurred. An alternate-key field might identify the terminal that initiated a transaction. To list all transactions for a given terminal in the order in which they occurred, generic positioning uses the field value of the desired terminal, and the file is then read sequentially.

Alternate Keys in a Relative File

You might use alternate keys in a relative 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-Key Files

For each primary structured file having one or more alternate keys, you must create at least one corresponding alternate-key file. An alternate-key file can be partitioned to span multiple volumes.

Each record in an alternate-key file refers to only one alternate key, but the file can contain references to more than one alternate key. Thus, with five alternate keys, the alternate-key file would have five records for each primary-file record (provided that none of the primary-file records contains a null value in any of the alternate-key fields). The primary file can also 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.

Here are some reasons you might want to have separate alternate-key files:

- A unique alternate key cannot share a file with other keys of different lengths.

- 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 fragments the file. With separate files, this fragmentation would not affect references to the other keys.

Here are two reasons why you do not want to have separate alternate-key files:

- 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 (FCB) is allocated for the first opening of an alternate-key file.

Figure 10 (page 34) illustrates the record structure of an alternate-key file. The length of a record in an alternate-key file is:

```
2 bytes for the key-specifier

+ the key-length of the longest alternate key included in
the record

+ the key-length of the associated primary key
```

The user defines the data file's primary key length if the data file is key sequenced. If it is a relative or entry sequenced data file, then its primary key length is 4 if it is format 1 or 8 if it is format 2.

Figure 10 Record Structure of an Alternate-Key File

Key Specifier (two bytes)	Alternate Key	Data File's Primary Key
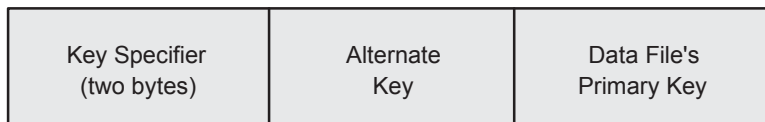
Note that if the primary-key file is audited, the alternate-key files also must be audited, unless keys are not automatically being updated. The primary-key length of an alternate-key file, as distinguished from the data file's primary key, depends on whether the file contains unique key references. 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, that key can be no longer than

```
2048 (maximum key-length of the alternate-key file for
key-sequenced files with increased limits)

- 2 for the key-specifier

- the key-length of the data file's primary key
```

Thus, if a data file's primary key is 33 bytes long, nonunique alternate keys within that file cannot be more than 2048 - 2 - 33 = 2013 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 2048 - 2 = 2046 bytes, regardless of the data file's primary key.

Alternate Keys and Record Locking

There are important considerations for alternate key locking particularly when aborting transactions.

This section describes Enscribe's handling of record locks for alternate key files and assumes a typical scenario in which an application opens a primary file and Enscribe internally opens the primary file's alternate key files but does not make those internal opens available to the application.

Record Locking Requests and Alternate Key Files

An application request to lock a record requires a file number as input. The file number is necessarily that of the primary file, since the application has no file number for any alternate key file.

If the request is FILE_READ[UPDATE]LOCK64_, READ[UPDATE]LOCK, FILE_LOCKREC64_ or LOCKREC, Enscribe locks the referenced primary file record but does not lock any alternate key records. There is no need to do so as long as the alternate key record is not being changed.

If the request is FILE_WRITEUPDATE[UNLOCK]64_/WRITEUPDATE[UNLOCK], Enscribe locks any affected alternate key records. For audited files, the locks are held until the transaction commits or aborts. For unaudited files, the locks are released when the application unlocks the corresponding primary record.

If the request is FILE_WRITE64_/WRITE and the file is audited, Enscribe locks inserted alternate key records and holds the locks until the transaction commits or aborts. If the request is FILE_WRITE64_/WRITE and the file is unaudited, Enscribe does not lock inserted alternate key records unless SETMODE 149,1 is in effect, in which case alternate key records are locked upon insertion and unlocked when insertion of the primary record and all of its alternate key records have completed.

Implementation of Updates to Alternate Key Records

Deletion/Insertion Sequence

If a null value is defined for an alternate key and an application updates that alternate key field from a non-null value to the null value, Enscribe deletes the corresponding alternate key record.

If a null value is defined for an alternate key and an application updates that alternate key field from the null value to a non-null value, Enscribe inserts a corresponding alternate key record.

If an application updates an alternate key field from one non-null value to another, Enscribe deletes the corresponding old alternate key record and inserts a new one. The deletion/insertion sequence is necessary because the alternate key value is part of the primary key of the alternate key file, and primary keys cannot be updated.

Transaction Aborts, Alternate Keys, and Locks

Backout of an aborted transaction executes asynchronously with respect to running applications. Because backout of alternate key insertions and deletions requires corresponding deletions and insertions which the system must protect with internally generated locks, transaction aborts can cause alternate key records to be locked and unlocked asynchronously with respect to an application's locking and unlocking of records in the primary file. In particular, it is possible for one process to have a primary record locked and for the system to lock a corresponding alternate key record on behalf of an aborting transaction of another process.

SETMODE 4 (Set Lock Mode) and Alternate Key Files

The SETMODE procedure call requires a file number as input. The file number is necessarily that of the primary file, since the application has no file number for any alternate key file.

Enscribe propagates SETMODE 4 to all alternate key files. The software also propagates SETMODE 4 to all secondary partitions of the primary file and alternate key files if any are partitioned.

SETMODE 4,6 and SETMODE 4,7 (Read through Lock with Warning)

Both SETMODE 4,6 and 4,7 allow FILE_READ64_, FILE_READUPDATE64_, READ and READUPDATE to read locked records. Such reads complete with warning code 9. With SETMODE 4,6 in effect, FILE_LOCKFILE64_, FILE_LOCKREC64_, LOCKFILE, LOCKREC, FILE_READ[UPDATE]LOCK64_, READ[UPDATE]LOCK, FILE_WRITE[UPDATE][UNLOCK]64_, and WRITE[UPDATE][UNLOCK] requests wait for lock release if the record or file in question is locked; with SETMODE 4,7, such requests are rejected with error 73.

SETMODE 4,6 and 4,7 can help applications to handle the asynchronous locking of alternate key records caused by transaction aborts. With SETMODE 4,6, a process whose write to a primary file collides with another process's system-generated lock of an aborting transaction will wait until the system-generated lock is released. This approach makes the lock contention invisible to the application in exchange for the possibility of an occasional lock wait. With SETMODE 4,7 in the same situation, a process can receive error 73 when writing to a primary record for which it holds a lock (because of a conflicting system-generated lock on an alternate key record). This approach avoids lock waits in exchange for the need to design the process to handle error 73 on write as retryable.

Relational Access

Relational access among structured files in a database is accomplished by obtaining a value from a field in a record in one file and using that value as a key to locate a record in another file.

Figure 11 (page 37) shows an example of relational access. All four of the files that are shown are primary data files. The illustrates a query operation in which the user wants to obtain information about all orders that were placed on 10-17-95.

In response to the user query command, the application program uses the date alternate-key field to locate the order summary records in the Order Header File that were placed on 10-17-95. For each applicable order summary record, the program:

- Uses the Name field value from the order summary record as an exact primary key to obtain the associated customer record from the customer file.

- Uses the Order Number field value from the order summary record as a generic primary key to obtain the associated line item records from the order detail file.

- Uses the Part Number field value from the line item records as exact primary keys to obtain the associated inventory record from the inventory file.

Figure 11 Relational Access Among Structured Files

Order Header File

Order No.	Name	Date	Total
0021	JONES, J.A.	10-17-95	
0022	ADAMS, A.B.	10-18-95	
0023	WATSON, N.A.	10-20-95	
0030	SMITH, C.J.	10-24-95	
0032	BROWN, E.G.	10-26-95	
0034	CARTER, J.M.	11-07-95	

Customer File

Name	Address	Region
ADAMS, A.B.	LOS ANGELES, CA	5
BROWN, E.G.	PITTSBURGH, PA	2
CARTER, J.M.	PLAINS, GA	3
JONES, J.A.	LAS VEGAS, NEV	4
SMITH, C.J.	BOSTON, MASS	1
WATSON, N.A.	CUPERTINO, CA	5

Order Detail File

Order No. /	Item No.	Part No.	Quantity	Item Total
0021	0001	0201	0001	
0021	0002	0310	0002	
0022	0001	0115	0006	
0023	0001	0205	0002	
0023	0002	0206	0025	

Inventory File

Part No.	Description	On-Hand	Price
0115	TOASTER	20	12.50
0205	PHONOGRAPH	52	55.00
0206	RADIO	210	9.95
0310	FRY PAN	19	37.50
0427	MIXER	97	69.95

3 System Procedures

File-System Procedures

You can use file-system procedures or a separate set of sequential I/O (SIO) procedures to create and access Enscribe files. The two sets of procedures are mutually exclusive; with regard to any given file, use one set or the other.

Table 4 (page 38) summarizes the functions of the applicable file-system procedures.

Table 4 File-System Procedures

Procedure	Description
AWAITIO[X]	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.
CONTROL_	Executes device-dependent operations on an open file.
DISK_REFRESH_	Writes information (such as the EOF pointer) in file control blocks (FCBs) to the associated physical disk volume. Supersedes REFRESH.
FILE_ALTERLIST_	Changes certain attributes (such as file type and file code) of a disk file that are normally set upon creation. Supersedes ALTER.
FILE_AWAITIO64_	Waits for completion of an outstanding I/O operation pending on an open file.
FILE_CLOSE_	Terminates access to a file and purges a temporary disk file. Supersedes CLOSE.
FILE_CONTROL64_	Executes device-dependent operations on an open file.
FILE_CREATE_	Creates a new structured or unstructured disk file. The file can be permanent or temporary. Supersedes CREATE.
FILE_CREATELIST_	Creates a new structured or unstructured disk file and specifies alternate-key information, partition information, or other attributes. Supersedes CREATE.
FILE_GETINFO_	Provides limited information about a file identified by file number. Supersedes FILEINFO, FILEINQUIRE, and FILERECINFO.
FILE_GETINFOBYNAME_	Provides limited information about a file. File is identified by name. Supersedes FILEINFO, FILEINQUIRE, FILERECINFO, DEVICEINFO, and DEVICEINFO2.
FILE_GETINFOLIST_	Provides detailed information about a file. File is identified by file number. Supersedes FILEINFO, FILEINQUIRE, and FILERECINFO.
FILE_GETINFOLISTBYNAME_	Provides detailed information about a file identified by file name. Supersedes FILEINFO, FILEINQUIRE, and FILERECINFO.
FILE_LOCKFILE64_	Locks an open disk file, making it inaccessible to other accessors.
FILE_LOCKREC64_	Locks a record (or a set of records if generic locking is enabled for a key-sequenced file) in an open disk file so that other processes cannot access it.
FILE_OPEN_	Establishes communication with a file and returns a file number. Supersedes OPEN.
FILE_PURGE_	Deletes a disk file that is not open. Supersedes PURGE.
FILE_READ64_	Following positioning, returns the first record of a subset; otherwise, it returns the next record in the current access path.
FILE_READLOCK64_	Is the same as FILE_READ64_, but it locks the record before reading it.

Table 4 File-System Procedures *(continued)*

Procedure	Description
FILE_READUPDATE64_	Returns the record indicated by the current key value; FILE_READUPDATE64_ is used to randomly read an open file.
FILE_READUPDATELOCK64_	Is the same as FILE_READUPDATE64_, but it locks the record before reading it.
FILE_RENAME_	Changes the name of an open file. If the file is temporary, causes the file to be made permanent. Supersedes RENAME.
FILE_RESTORE_POSITION_ FILE_SAVEPOSITION_ FILE_SETKEY_	Sets position by primary or by alternate key within key-sequenced, entry-sequenced, relative, and queue files; defines a subset of the file for subsequent access by setting the current position, access path, and positioning mode. This procedure expects the primary keys for relative and entry-sequenced files to be 8 bytes long. Supersedes KEYPOSITION.
FILE_SETMODENOWAIT64_	Sets device-dependent functions for an open file.
FILE_SETPOSITION_	Sets position by primary key within an entry-sequenced or relative 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 within an unstructured file. Procedure accepts an 8-byte record specifier so it can work with format 2 files. Supersedes POSITION.
FILE_UNLOCKFILE64_	Unlocks an open disk file currently locked by the caller; unlocks any records in the file that are currently locked by the caller.
FILE_UNLOCKREC64_	Unlocks a record currently locked by the caller. If generic locking is enabled, calls to FILE_UNLOCKREC64_ are ignored.
FILE_WRITE64_	Inserts (adds) a new record into an open disk file location positioned by the last call to READ[X], FILE_READ64_, READUPDATE[X] or FILE_READUPDATE64_.
FILE_WRITEUPDATE64_	Replaces (updates) or deletes data in the record indicated by the current key value of an open file.
FILE_WRITEUPDATEUNLOC K64_	Is the same as FILE_WRITEUPDATE64_, but unlocks the record after its contents are updated or deleted.
FILEERROR	Helps determine whether a failed call should be retried.
FILENAME_FINDNEXT_	Returns the next name in a set of named entities. The set is defined in a call to FILENAME_FINDSTART_. Supersedes GETDEVNAME and NEXTFILENAME.
FILENAME_FINDNEXT64_	Returns the next name in a set of named entities. The set is defined in a call to FILENAME_FINDSTART_. Supersedes GETDEVNAME and NEXTFILENAME.
FILENAME_FINDSTART_	Sets up a search of named entities.
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.
FNAMEEXPAND	Expands an external file identifier to internal form.
KEYPOSITION	Sets position by primary key within a key-sequenced or queue file or by alternate key within key-sequenced, entry-sequenced, and relative files; defines a subset of the file for subsequent access by setting the current position, access path, and positioning mode. This procedure expects the primary keys for relative and entry sequenced files to be 4 bytes long.
LOCKFILE	Locks an open disk file, making it inaccessible to other accessors.

Table 4 File-System Procedures *(continued)*

Procedure	Description
LOCKREC	Locks a record (or a set of records if generic locking is enabled for a key-sequenced file) in an open disk file so that other processes cannot access it.
POSITION	Sets position by primary key within an entry-sequenced or relative 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 within an unstructured file. Procedure accepts a 4-byte record specifier; it cannot be used with files larger than 4 GB.
READ[X]	Following positioning, returns the first record of a subset; otherwise, it returns the next record in the current access path.
READLOCK[X]	Is the same as READ[X], but locks the record before reading it.
READUPDATE[X]	Returns the record indicated by the current key value; READUPDATE[X] is used to randomly read an open file.
READUPDATELOCK[X]	Is the same as READUPDATE[X] except that it locks the record before reading it.
REPOSITION	Restores the disk file position information saved with a previous SAVEPOSITION call.
SAVEPOSITION	Saves the current disk 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 for an open file.
UNLOCKFILE	Unlocks an open disk file currently locked by the caller; unlocks any records in the file that are currently locked by the caller.
UNLOCKREC	Unlocks a record currently locked by the caller. If generic locking is enabled, calls to UNLOCKREC are ignored.
WRITE[X]	Inserts (adds) a new record into an open disk file location positioned by the last call to READ[X] or READUPDATE[X].
WRITEUPDATE[X]	Replaces (updates) or deletes data in the record indicated by the current key value of an open file.
WRITEUPDATEUNLOCK[X]	Is the same as WRITEUPDATE[X], but unlocks the record after its contents are updated or deleted.

Procedure Call Completion

If a file is open with nowait I/O specified, calls to CONTROL, FILE_CONTROL64_, FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, FILE_WRITE64_, FILE_WRITEUPDATEUNLOCK64_, FILE_SETMODENOWAIT64_, LOCKFILE, LOCKREC, READ[X], READLOCK[X], READUPDATE[X], READUPDATELOCK[X], UNLOCKFILE, UNLOCKREC, WRITE[X], WRITEUPDATEUNLOCK[X], or SETMODENOWAIT must be completed by a corresponding call to AWAITIO[X]/FILE_AWAITIO64_.

If a file is open with nowait I/O specified, calls to FILE_RENAME_, FILE_SETKEY_,FILE_SETPOSITION_,KEYPOSITION, POSITION, REPOSITION, SETMODE, FILE_SETMODENOWAIT64_ or SETMODENOWAIT are rejected with file-system error 27 if there are any outstanding operations pending. Regardless of whether the file was opened with waited or nowait I/O specified, a return from a call to CANCELREQ, FILE_CLOSE_, FILE_CREATE_, FILE_GETINFO_, FILE_GETINFOBYNAME_, FILE_GETINFOLIST_, FILE_OPEN_ (unless `options` .<1> = 1), FILE_PURGE_, FILE_RENAME_,

FILENAME_FINDNEXT64_, FILENAME_FINDNEXT_, KEYPOSITION, POSITION, or SETMODE indicates a completion.

File Number Parameters

All of the file-system procedures except DISK_REFRESH_, FILE_CREATE_, FILE_GETINFOBYNAME_, FILE_OPEN_, FILE_PURGE_, FILENAME_FINDNEXT64_, and FILENAME_FINDNEXT_, use the `filenum` parameter returned by the FILE_OPEN_ procedure to identify which file the call refers to. The FILE_CREATE_, FILE_GETINFOBYNAME_, FILE_OPEN_, and FILE_PURGE_ procedures refer to the file by `filename`; the LASTRECEIVE and FILE_REPLY64_, REPLY procedures always refer to the $RECEIVE file. For every procedure except FILE_OPEN_ and AWAITIO[X]/FILE_AWAITIO64_ that has a`filenum` parameter, 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 nowait file. The tag is passed back to the application process, through the AWAITIO[X]/FILE_AWAITIO64_ 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

The buffer parameter in a file-system procedure call specifies where the data is to be read from or written to.

For I/O operations such as FILE_READ64_, READ, FILE_WRITE64_ or WRITE, the designated buffer must be of the type integer (INT) or double-integer (INT(32)) and it must reside in the user's data area (P-relative read-only arrays are not permitted).

For extended I/O operations such as READX or WRITEX, the designated buffer must be of type INT or INT(32) and it can reside in either the user's data area or an extended data segment (P-relative read-only arrays are not permitted).

Transfer Count Parameter

The read-count or write-count parameter in a file-system procedure call specifies how many bytes are to be read or written.

A SETMODE procedure call with a function code of 141 enables and disables bulk transfers of data between an extended data segment (or the upper 32 K of the data stack) and a DP2 disk file that has been opened for unstructured access. Note that the SETMODE 141 call requires that the file be opened for unstructured access (bit 2 of open^flags = 1) and for exclusive access (bits 10 and 11 of open^flags = 2).

With the bulk transfer feature disabled, you can transfer from 0 to 4096 bytes in a single operation.

With the bulk transfer feature enabled, you can transfer up to 30 K bytes in a single operation. The amount of data transferred must be a multiple of 2 K bytes. Note that with the bulk transfer feature enabled, the only data transfer I/O operations that are allowed are READX, READUPDATEX, WRITEX, and WRITEUPDATEX.

Condition Codes

Some file-system procedures return a condition code indicating the outcome of the operation. For these procedures, 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 FILE_GETINFO_ procedure to determine the error).

> (CCG)   A warning message was generated (typically EOF, but see the individual procedures for the meaning of CCG or call FILE_GETINFO_ to get an error number).

= (CCE)   The operation was successful.

Error Numbers

An error number is associated with each call completion. As shown in Table 5, the error numbers fall into three major categories. The setting of the condition code indicates the general category of the error associated with a completed call.

Table 5 Error Number Categories

Error Number	Condition Code	Category
0	= (CCE)	*No error*. The operation executed successfully.
1-9	> (CCG)	*Warning*. The operation executed with the exception of the indicated condition. For warning 6 (system message received), data is returned in the application process buffer.
10-255, 512-32767	< (CCL)	*Error*. The operation encountered an error or a special condition (such as a transaction abort for an audited file) that the application must recognize. For data transfer operations, either none or part of the specified data was transferred (except data communication error 165, which indicates a normal completion with data returned in the application process buffer).
300-511	< (CCL)	*Application-defined error*. These error numbers are reserved for use by application processes.

You can obtain the error number associated with an operation on an open file using the FILE_GETINFO_ system procedure and passing the file number of the file in error:

```
status := FILE_GETINFO_ ( filenum, lasterror ) ;
```

The function value returned to status indicates the success of the call to FILE_GETINFO_. Usually this value is 0 unless a programming error is made, such as supplying an invalid file number. The error associated with the preceding operation (such as FILE_READ64_, READ, FILE_WRITE64_ or WRITE) is returned to the second parameter (lasterror).

You can obtain the error number of a preceding AWAITIO[X]/FILE_AWAITIO64_ on any file or of a waited FILE_OPEN_ that failed, using FILE_GETINFO_ with file number -1:

```
status := FILE_GETINFO_ ( -1, lasterror ) ;
```

Note that if the FILE_OPEN_ procedure fails, it sets the filenum parameter to -1.

Similarly, you can get the error number of a preceding FILE_CREATE_ or FILE_PURGE_ operation that failed using FILE_GETINFO_ with filenum = -1.

File Access Permissions

The disk file must be open with read or read/write access for a FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X], or READUPDATELOCK[X] call to be successful; if not, the call is rejected with an error 49 (access violation).

The disk file must be open with write or read/write access for CONTROL, FILE_CONTROL64_, FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITE[X], WRITEUPDATE[X], or WRITEUPDATEUNLOCK[X] calls to be successful; if not, the call is rejected with an error 49 (access violation).

The caller must have purge access to the disk file if FILE_PURGE_ or FILE_RENAME_ calls are to be successful; otherwise the call is rejected with an error as 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, however, are declared externally to the application program in a system file named $SYSTEM.SYSTEM.EXTDECS0. 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.EXTDECS0 (external-procedure, ...)
procedure-declarations
```

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

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 ( KEYPOSITION, WRITEUPDATE,
?            FILE_OPEN_, FILE_CLOSE, READ, WRITE, POSITION, )
```

compiles only the external declarations for the KEYPOSITION, WRITEUPDATE, FILE_OPEN_, FILE_CLOSE_, READ, WRITE, and POSITION procedures.

Sequential I/O (SIO) Procedures

The sequential I/O (SIO) procedures summarized in Table 6 provides HPE Tandem Application Language (TAL) programmers with a standardized set of procedures for performing common I/O operations. These operations include reading and writing IN and OUT files, and BREAK from a terminal. These procedures are primarily intended for system 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 SIO procedures also contain a set of DEFINEs and LITERALs that allocate control-block space, specify file-opening characteristics, and both set file and check file-transfer characteristics.

Table 6 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

SIO procedures have these characteristics:

- 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. Default characteristics are set to facilitate their most general use.

- Error recovery is automatic. Each fatal error causes a comprehensive error message to be displayed, all files to be closed, and the process to be aborted. 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 is 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.

- They can be used with the INITIALIZER procedure to make run-time changes. File transfer characteristics, such as record length, can be changed using the operating system Command Interpreter's ASSIGN command.

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

For a thorough discussion of the SIO procedures, refer to the *Guardian Programmer's Guide.*

4 General File Creation and Access Information

This chapter describes general file creation and access information for all Enscribe file types.

File Creation

This subsection discusses certain parameters that you can specify and capabilities that you can enable when creating Enscribe files.

File Codes

When creating an Enscribe disk file, you can assign it an arbitrary numeric file code. This code is typically used to categorize files according to the kind of information they contain.

File codes 100 through 999 are reserved; using them causes unpredictable results.

If you do not specify the file code, it defaults to 0.

Disk Extent Sizes

When you create an Enscribe file, you can specify the maximum amount of physical disk space to be allocated for that file. Physical space is allocated in the form of extents. An extent is a contiguous block of disk space that can range in size from a single page (2048 bytes) to 65,535 pages (134,215,680 bytes) for format 1 files or to 536,870,912 pages for format 2 files.

For EKS files with increased limits, the primary partition has a primary extent size greater than or equal to 140 pages. Each secondary partition has its own primary and secondary extent size.

By default every Enscribe disk file and partition has 16 extents; you can specify that a particular nonpartitioned file or the partitions of a key-sequenced file have more than that number.

The first extent is called the primary extent, and its size can differ from the secondary extents. All of the secondary extents for a given file or partition are the same size as one another.

If you do not specify the primary and secondary extent sizes during file creation, they both default to one page.

The extent size must be an integral multiple of either the buffer size (for unstructured files) or the block size (for structured files); the buffer size or block size, in turn, must be 512 bytes, 1 KB, 2 KB, or 4 KB. 32 KB block size is only supported for key-sequenced files with increased limits. The file system automatically rounds up the specified extent size, if necessary, to enforce this requirement.

The file system also rounds up any extent size specified as an odd number of pages if the buffer size or block size is 4096. Therefore, if you want to have a file with extent sizes consisting of an odd number of pages, the buffer size or block size must be 2048 or less.

In addition, you can supply a value called MAXEXTENTS that specifies the maximum number of extents (16 or greater) to be allocated for the file. If you do not specify the maximum number of extents, it defaults to 16.

For partitioned files other than key-sequenced files, MAXEXTENTS is always 16 per partition. For partitioned key-sequenced files, you can specify a MAXEXTENTS parameter greater than 16.

For non-partitioned files or partitioned key-sequenced files, you can also change the MAXEXTENTS value dynamically during program execution by either issuing a CONTROL 92 procedure call or supplying a MAXEXTENTS parameter in a FUP ALTER command.

File Formats Supported: Format 1 and Format 2

The D46 release supports larger partitions then were previously supported. This change allows files and file partitions to increase from the format 1 size of 2 gigabytes to the format 2 size of 1

terabyte (although the upper limit of the actual file size depends upon the size of the largest single disk).

Both file formats are supported. The user usually does not have to specify the format. If left unspecified, the system chooses format 1.

File format 2 is required when:

- An enhanced key-sequenced (EKS) file is used. EKS files are supported in J06.11 and H06.22 and later RVUs.

- A key-sequenced file with increased limits is used. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with specific SPRs. For a list of the required H06.28/J06.17 SPRs, see SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release (page 18).

- An unpartitioned file (or a partition) has a maximum size that is over 2 GB – 1 MB.

- A non-key-sequenced file has a maximum size that is 4 GB or more (counting all partitions).

- An extent size is over 65535.

Users should take advantage of the properties of format 2 when appropriate.

Table 7 Comparison of Format 1 Versus Format 2 Files

File Attribute or Procedure Call	Format 1 Files	Format 2 Files
Block Size	512 bytes to 4 KB	512 bytes to 32 KB
Maximum Partitioned File Size (non-key-sequenced)	4 GB - 4 KB	16 TB
Maximum Partitioned File Size (key-sequenced)	4 GB - 4 KB	LKS2: 16 TB EKS with increased limits: ~127 TB
Maximum Unpartitioned File Size (single partition)	2 GB - 1 MB	~1 TB
Maximum Record Size: Entry-Sequenced, Relative, Key-Sequenced (LKS or EKS). Queue files have the same record sizes as Key-Sequenced files.	Entry-Sequenced block size - 24 Relative block size - 24 Key-Sequenced block size - 34	Entry-Sequenced block size - 48 Relative block size - 48 Key-Sequenced block size - 56 (except for 32 KB in which case it is 27648)
Extent Sizes (pages)	64 K - 1 pages (128 MB)	Using new 32-bit fields, extent sizes can be larger than format 1 files. (The limit is dictated by the largest size of a single physical disk).
Maximum Number of Extents	Same (978 or less)	Usable value is smaller than format 1 limit (928 or less)
End-of-File Values[1]	514 byte disks = 16 extents of 65504 pages = EOF of 2,146,435,072 bytes 512 byte disks = 16 extents of 65492 pages = EOF of 2,146,041,856 bytes	May be larger than standard
Number of Partitions	16 partitions	Non-Key-Sequenced: 16 partitions LKS2: 16 partitions EKS with increased limits: 128 partitions

Table 7 Comparison of Format 1 Versus Format 2 Files *(continued)*

File Attribute or Procedure Call	Format 1 Files	Format 2 Files
File Types	All	All
Non-Key-Sequenced Primary Key Value in Alternate Key Records	4-byte (32-bit) primary-key value	8-byte (64-bit) primary-key value

[1] For unstructured format 1 files in 512 byte disks such as ESS/JBOD, the pri-extent-size and sec-extent-size must both be divisible by 14. If the file is unpartitioned, DP2 automatically rounds up the size. The maximum page size that is divisible by 14 is 65492, therefore the EOF (16 extents of 65492 pages) is 2,146,041,856 bytes.

Partition and File Format Compatibility

All partitions of a file are created with the same file format version (format 1 or 2).

For non-key-sequenced files, a partition that is created independently must have the same format as all the other partitions of the file. If that partition does not have the same format, it cannot be opened and a warning message of FEPARTFAIL is displayed.

Key-sequenced files with a mixture of partition formats, providing they meet all other compatibility requirements, are accessible and can be opened. This should assist in the process of converting large partitioned key-sequenced format 1 files to large partitioned key-sequenced format 2 files.

NOTE: Partition compatibility requires the same record and block sizes in all partitions. This has consequences for some file types. In particular, for key-sequenced files with some record/block size combinations, it may not be possible to have a mixture of formats among the partitions.

File Size Limits

Applications can encounter problems if the files it uses are too large. The following paragraphs discuss limits and considerations for the different Enscribe file types.

Files Secured With Enhanced File Privileges

When format 1 Enscribe files are secured using the new OSS file privilege attributes, those files are limited to approximately 953 secondary partitions (file label size of 3960 bytes). These new file privilege attributes cannot be set on format 1 Enscribe files that have approximately 945 partitions (a file label size of 3926 bytes or greater).

These restrictions also apply to format 1 partitioned files. Format 2 Enscribe files are not affected.

Users with files affected by this limitation may reorganize the files to create larger primary and secondary extents. Larger primary and secondary extents result in a lesser number of extents for the same file. The following command lists the attributes of a file including the "File Label" size:

```
fup info <filename>, detail
```

Use the following commands to create a new file with larger primary and secondary extent sizes:

```
fup dup <filename>, <new-filename>, ext (pri-ext-size), sec-ext-size)
```

Partition Limit

A single format 1 partition (or an unpartitioned format 1 file) is limited to a maximum of 2 GB minus 1 MB. If you try to create a file or partition with extent sizes and maxextents that specify a potential greater than this amount, then a format 2 file will be created (if the default system format selection is in effect). If 2 GB minus 1 MB is exceeded when format 1 is explicitly specified, or if the size exceeds 1 terabyte, then error 583 or 21 is returned.

Key-Sequenced and Queue Files

Legacy key-sequenced files can have up to 16 partitions and the maximum possible size is ~16 TB. EKS key-sequenced files without increased limits can have up to 64 partitions and the

maximum possible size is ~63 TB. However, key-sequenced files greater than 2 GB may encounter problems if programs use the 32-bit file-system attributes that describe them. In particular, the End of File (EOF) and Maximum File Size attributes cannot describe files greater than approximately 2 GB (as a signed integer) or 4 GB (unsigned) in the 32-bit form, so the 64-bit forms should be used instead.

EKS key-sequenced files with increased limits can have up to 128 partitions and the maximum possible size is ~127 TB.

Queue files cannot be partitioned.

Other File Types

Entry-sequenced and relative files can have up to 16 partitions. However, the system imposes a limit of approximately 4 GB (4 GB minus 4 KB) on these types of files if they are format 1. The system returns an error 21 if there is an attempt to exceed the limit.

Files in the Range of 2 GB to 4 GB

With files between 2 GB and 4 GB, applications might encounter problems with arithmetic operations that involve signed 32-bit integers. The 32-bit values that are used by some file-system attributes (such as EOF) must be interpreted as unsigned integers to prevent errors. Use of the 64-bit forms of these attributes instead is recommended.

Audit-Checkpoint Compression

When an Enscribe file is being audited by TMF or has been opened with a sync depth greater than 0, updating a record in the particular file causes an audit-checkpoint (AC) record to be created. The AC record, which describes changes incurred as a result of the update, is sent to the backup disk process. If the file is being audited by TMF, then the AC record also is sent to the audit trail's disk process and eventually written to the audit trail.

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

An AC record includes:

- a record header approximately 64 bytes long

- a copy of the record before the update, or before-image

- a copy of the record after the update, or after-image.

If a data record is 1000 bytes long, the AC record for its update would be 2064 bytes long.

AC compression shortens parts (2) and (3) of the AC record by omitting the unchanged fragments of the data record. The AC record must include the record key (in a key-sequenced or queue file), the relative byte address (in an unstructured or entry-sequenced file), or the record number (in a relative 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:

- Processor and memory cycles are reduced during message transferal, although some extra cycles are required to perform the compression.

- 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.

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 disk process does not compress the record. This limit is subject to change from release to release.

- If an update changes the record length, the disk 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 number of fragments are changed, the entire remainder of the record is considered to be 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 and queue records with large keys reduce the effectiveness of AC compression because the key must be kept in the AC record.

When creating a file with the FILE_CREATE_ procedure, you enable or disable auditcheckpoint compression by setting options bit 13 to 1 or 0, respectively.

When creating a file with FUP, you enable or disable audit-checkpoint compression by specifying AUDITCOMPRESS or NO AUDITCOMPRESS, respectively, in the FUP CREATE command.

When you open a file, the audit-compression feature is enabled or disabled at that time depending upon what was specified for the file when it was created. During program execution, you can enable or disable this feature dynamically by:

- Specifying AUDITCOMPRESS or NO AUDITCOMPRESS, respectively, in a FUP ALTER command, or by

- Specifying param1 = 1 or 0, respectively, in a SETMODE 94 procedure call.

When you do so, however, whatever you designate only applies until you reverse it with a subsequent FUP ALTER command or SETMODE 94 call or until the file is closed.

Write Verification

When creating Enscribe disk files, you can enable the write verification feature. This feature ensures the integrity of each subsequent write operation to that file. With write verification enabled, the just-written data is read back from the disk and compared with the corresponding data in the memory of the CPU. Note, however, that this requires an additional disk revolution.

When creating a file with the FILE_CREATELIST_ procedure, you enable or disable write verification by setting item code 73 to 1 or 0, respectively.

When creating a file with FUP, you enable or disable write verification by specifying VERIFIEDWRITES or NO VERIFIEDWRITES, respectively, in the FUP CREATE command.

When you open a file, the write verification feature is enabled or disabled at that time depending upon what was specified for the file when it was created. During program execution, you can enable or disable this feature dynamically by:

- Specifying VERIFIEDWRITES or NO VERIFIEDWRITES, respectively, in a FUP ALTER command, or by

- Setting bit 15 of the param1 parameter to 1 or 0, respectively, in a SETMODE 3 procedure call.

When you do so, however, whatever you designate only applies until you reverse it with a subsequent FUP ALTER command or SETMODE 3 call or until the file is closed.

File Access

The following paragraphs describe different ways to access Enscribe files.

Opening and Closing Files

You use the FILE_OPEN_ procedure to establish communication with a permanent disk file:

```
LITERAL name^length = 23,
        syncdpth = 1;
LITERAL usebigkeys = 1D;

INT error;
INT filenum;
STRING .filename [0:22] := "$VOL1.MYSUBVOL.DATAFILE";
error := FILE_OPEN_(filename:name^length,
filenum,,,,syncdpth,                  ! read-write access,
,,,,usebigkeys);                      ! shared access,
                                      ! waited I/O,
                                      ! syncdepth = 1
                                      ! use 64-bit primary key
```

NOTE: With the release of D46, users may now use 64-bit primary keys instead of 32-bit values for unstructured, relative or entry-sequenced disk files. If the default 32-bit keys are used, only files up to 4 GB can be opened. If the 64-bit selection is used, access to any size file is allowed, but 32-bit interfaces such as POSITION must be avoided.

Once the file is open, use the file number (FILENUM) returned by the FILE_OPEN_ procedure to identify the file to other file-system procedures.

Use the FILE_CLOSE_ procedure to terminate access to a disk file:

```
error := FILE_CLOSE_(filenum);
```

If you do not explicitly close the file, the file remains open until the process stops, at which point all files are automatically closed.

To establish communication with a temporary file, use the temporary file name returned by the FILE_CREATE_ procedure:

```
LITERAL temp^name^len = 256;
INT namelen;
INT error;
INT temp^filenum;
STRING .temp^filename [0:temp^name^len-1];
temp^filename ':=' "$VOL";
namelen := 4;
error := FILE_CREATE_ (temp^filename:temp^name^len, namelen);
error := FILE_OPEN_ (temp^filename:namelen,
                     temp^filenum);
```

FILE_CREATE_ returns a file name that you can use when you call FILE_OPEN_.

Temporary files are purged when you close them. If you do not want a temporary file to be purged, call FILE_RENAME_ using the file number returned from FILE_OPEN_ to make the file permanent.

Opening Partitioned Files

When you open the primary (first) partition of a partitioned file, all of the associated secondary partitions are also opened automatically. If a secondary partition cannot be opened, access to the file is still granted but the FILE_OPEN_ procedure returns a code 3 warning indication and some operations might not work. You can call the FILE_GETINFOLIST_ procedure to identify the highest numbered partition that did not open.

Individual partitions cannot be opened separately unless you specify unstructured access in the FILE_OPEN_ call.

Read Reverse With Structured Files

You can read Enscribe structured files (key-sequenced, queue, entry-sequenced, and relative) sequentially in descending key or record number order. You enable this feature by setting a bit in the options parameter of FILE_SETKEY_ or the positioningmode parameter of the KEYPOSITION procedure call.

Read Reverse and Position-to-Last Feature

Normally the KEYPOSITION procedure resets the file pointers so that they point to the first record that satisfies the positioning criteria specified by the key-value, key-len, compare-len, and positioning-mode parameters. When reading a file in reverse order, however, you probably want to point to the last record in the file that satisfies the positioning criteria. You enable the position-to-last feature by setting a second bit in the positioning-mode parameter of the KEYPOSITION procedure call.

For example, consider the record sequence:

Record Number	Key Value
0	AAA
1	ABA
2	ABB
3	ABC

Following an approximate KEYPOSITION to key-value AB with a specified key-length of 2 and read-reverse enabled, a call to FILE_READ64_/READ would return record number 1 from the set of records shown in the preceding example. The same call to KEYPOSITION, but with position-to-last enabled by the positioning-mode parameter, would instead result in record number 3 being returned by the FILE_READ64_/READ call.

For the primary key of entry-sequenced and relative files, the key value parameter to KEYPOSITION is a 4-byte string containing a doubleword record number value. When using read reverse and approximate positioning in conjunction with those types of files, initial positioning is performed to the first record whose record number is equal to or less than the record number passed in key-value. Records are then returned in descending record number order by successive calls to FILE_READ64_/READ. The position-to-last option is ignored for KEYPOSITION operations to an exact record number in an entry-sequenced or relative file.

Positioning to the last record in a file with KEYPOSITION is accomplished by specifying a key length of 0 and specifying approximate positioning, read-reverse, and position-to-last in the positioning-mode parameter. Following such a KEYPOSITION operation, the subsequent call to FILE_READ64_/READ will return the last record in the file.

Read Reverse and SAVEPOSITION

Because of the read reverse feature, the SAVEPOSITION system procedure requires a 7-word positioning buffer to save the current position in an entry-sequenced or relative file that has no alternate keys.

Only four words of the positioning buffer are used when read forward positioning is in effect. Read reverse has no impact on the required size of the positioning buffer for key-sequenced files, queue files, entry-sequenced files with alternate keys, and relative files with alternate keys.

Read Reverse and the Record Pointers

For entry-sequenced and relative files, the state of the current-record and next-record file pointers following a FILE_READ64_, READ, FILE_WRITE64_ or WRITE are:

Current-record pointer = record number or address of the last record read or written.

Next-record pointer = record number or address that follows the current-record pointer.

For entry-sequenced files, the record pointers contain a record address; for relative files, the record pointers contain a relative record number. The contents of the current-record and next-record pointers are accessible with the FILE_GETINFOLIST_ system procedure.

Following a call to FILE_READ64_/READ when reverse-positioning mode is in effect, the next-record pointer contains the record number or address that precedes the current record number or address.

Following a read of the first record in a file (where the current-record pointer is 0) when reverse-positioning mode is in effect, the next-record pointer contains an invalid record number or address because no previous record exists. In such a case, a subsequent call to FILE_READ64_/READ will result in a CCG completion with file-system error code 1 (EOF) and a call to FILE_WRITE64_/WRITE results in a file-system error code 550 (illegal position) because an attempt was made to write beyond the beginning of the file.

File Expiration Dates

Each file has an associated expiration date value in the form of a 4-word timestamp. For purge operations this value is checked against the current time; if it is later than the current time, the purge is disallowed and a 1091 error code is returned. This check is in addition to, and separate from, the usual purge authority checking. You can set the expiration date of a file by using the FILE_ALTERLIST_ system procedure with an item code of 57. The initial expiration date value when a file is created is zero, which represents a date far in the past, so all files are initially purgeable as far as the expiration date is concerned.

NOTE: Any Enscribe files existing before the C10 software release are presumed to have a zero expiration date.

When you change the expiration date of a file, it is not changed for any associated alternate-key files but is changed for the secondary partitions of a partitioned file (unless the partonly parameter was set to 1 in the FILE_ALTERLIST_ call). You must have read and write authority to change the expiration date of a file. In addition, you cannot set an expiration date for a temporary file because such files are automatically purged when you close them.

File Creation and Last-Opened Timestamps

Each file has associated with it a creation date value and a last-opened date value in the form of 4-word timestamps.

You can obtain the creation date or last-opened date of a file by issuing a FILE_GETINFOLIST_ procedure call.

Using CONTROL 27 to Detect Disk Writes

For nonpartitioned files, you can use the CONTROL/FILE_CONTROL64_ system procedure with a function code of 27 to detect when a disk write operation (FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITE, WRITEUPDATE, or WRITEUPDATEUNLOCK) completes for the file designated by the specified file number.

This procedure call is useful in application environments where several different processes are accessing the same database files and where you must quickly determine that the content of certain critical records has changed.

Note that a CONTROL 27 call completion does not guarantee that any data in the file has actually changed; it merely indicates that a disk write has completed against the file and it is a reasonable time to examine the critical record(s) for new data.

To assure that no updates are missed as you read a record, first issue a nowait CONTROL 27 call against the file through one file open, read the record through another file open, and then check for a completion of the CONTROL 27 call. If you issued the CONTROL 27 call after reading

the record, a disk write by another process may have occurred between your read call and the CONTROL 27 call.

Because the automatic resending of a CONTROL 27 request could be queued behind a disk write request by another process (thus missing it), the file open through which you issue the CONTROL 27 call should have a sync depth of 0. In such a case, you should treat path errors (200-211) and network errors (240-249) as successful completions in that you then immediately check the critical record(s) for new data.

Using Cache Buffering or Sequential Block Buffering

The Enscribe product provides two buffer management options, cache buffering and sequential block buffering, that you can sometimes use to make I/O operations more efficient.

Cache Buffering

The cache, or buffer pool, is an area of main memory reserved for buffering blocks read from disk. You use the PUP SETCACHE command to specify the cache size.

When a process reads a record, the software 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 disk into the cache and 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, some other block must be selected to be overlaid. A choice of three different cache access types is available: sequential access, random access, and system managed.

When a process writes a record, what happens differs according to the options selected when the file was opened. If the buffered cache feature is not used, the cache block that contains the record is modified, and then immediately written to disk. If the block to be modified is not in the cache, it is first read from the disk. The modified block remains in cache, however, until the buffer space is needed for overlay; this is called write-through cache.

You can open a file with buffered cache so that the cache contents are written to disk, 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 disk is required. Database changes, however, do not get into the actual disk 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.

- TMF forces flushing.

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

- The cache configuration is changed.

- The buffer size of an unstructured file is changed by a SETMODE procedure call, causing all cache buffers for that file to be flushed.

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

- When the disk 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.

Any file can be either buffered or write-through; the defaults are buffered for audited files and write-through for non audited files.

Write-through cache is the default for nonaudited files, because a system failure or unplanned disk process takeover (with a sync-depth of 0) could cause the loss of buffered updates and an application might not detect or handle such a loss properly. Such a loss of buffered updates

would be indicated by error 122. Buffered cache is the default for audited files because TMF can recover committed, buffered updates lost due to a system failure.

NOTE: Be careful in using the combination of nonaudited, buffered files with a sync-depth of 0 (no checkpointing). This combination provides high-performance updates but might compromise data integrity in certain situations. Restartable applications (old master to new master, for example) are not a concern. However, with online transaction applications there is a risk that some updates buffered in the cache could be lost if there is a primary CPU failure. This is not a problem if the primary-to-backup switch is deliberately set or is due to a controller path error, because no processor failure is involved. If there is a disk process backup takeover due to primary CPU failure, the disk process returns an error 122 on the next request of any kind for that file, indicating a possible prior loss of buffered updates.

If a nonaudited buffered file with a sync-depth of 0 is used, the application should use SETMODE 95 to flush the buffered updates to disk before closing the file. The application should not depend on the FILE_CLOSE_ procedure to do this, because FILE_CLOSE_ does not return an error and consequently there would be no indication of a possible prior loss of buffered updates (error 122).

The disk process avoids fragmentation of cache memory space by grouping all 4096- byte blocks in one area, all 2048-byte blocks in another area, and so forth. You set the amount of cache memory devoted to each block size by using the PUP SETCACHE command. For the system disk, the system operator sets the amount of cache memory devoted to each block size during system configuration.

The disk process cache manager maintains ordered lists of its cache blocks (one for each size of cache block), with the most recently used at the top of the list and the least recently used at the bottom. When the cache manager needs a new block, it typically uses the entry at the bottom of the appropriate list. After a block has been used, its entry moves to the top of the list and it thereby becomes the most recently used block. As blocks are used, the various entries in the list gradually migrate downward toward the bottom of the list.

Index and bit-map blocks, however, are kept longer in cache than are data blocks. The cache manager always uses data blocks whenever they reach the bottom of the list, but allows index and bit-map blocks to migrate through the list twice before using them.

Because this technique is believed to be advantageous in every application environment, there is no way to disable it.

Sequential Block Buffering

When reading a file sequentially, you can reduce system overhead if you enable sequential block buffering when you open the file. Note that this applies to read access only.

Sequential block buffering essentially relocates the record deblocking buffer from the disk process to your application's process file segment (PFS). The software then uses the PFS buffer to deblock the file's records.

Without sequential block buffering, the file system must request each record separately from the disk process; for each record, this involves sending an interprocess message and changing the environment. With sequential block buffering enabled, an entire block is returned from the disk process and stored in the PFS buffer. Once a block is in the PFS buffer, subsequent read access to records within that block is performed entirely by the file system (not the disk process) and requires no hardware disk accesses, no communication with the disk process, and no environment changes.

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.

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

- The block has been traversed, at which time the disk process fetches another block.

- An intervening FILE_SETPOSITION_, FILE_SETKEY_, POSITION, or KEYPOSITION is performed. In this case, the next READ request causes a new block to be fetched.

- A disallowed request (such as FILE_READUPDATELOCK64_, FILE_READLOCK64_, FILE_READUPDATE64_, READUPDATELOCK, READLOCK, READUPDATE, FILE_LOCKREC64_, LOCKREC, or a write request)

Considerations

Note that sequential block buffering 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, because they always involve the disk process, write operations automatically clear the buffer.

Sequential block buffering ignores any record locks that are currently in effect for the records in the block. Sequential block buffering does not, however, bypass a file lock when the block is first retrieved from the disk process.

To change a record that has been read from a block buffer, you should first perform intervening FILE_SETPOSITION_, FILE_SETKEY_, POSITION, KEYPOSITION, FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK, or READUPDATELOCK operations to fetch the record. Doing so:

- Ensures that the record has not been altered or deleted by another user since the block was read.

- Ensures that the record is not currently locked by another process.

- Locks out other processes from the record to be updated.

FILE_OPEN_ Parameters

The FILE_OPEN_ procedure's sequential-block-buffer and buffer-length parameters govern creation of the buffer.

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.

The buffer-length parameter is the more significant parameter because:

- If it is zero, absent, or longer than the space available in the PFS, the open operation succeeds but returns an indication with warning 5 (failure to provide sequential buffering) and block buffering is not used.

- If it is greater than the file's block size, the buffer will be created with the specified size.

- If it 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 a buffer-length parameter of 128, the buffer will be created for a block size of 4096.

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, open the primary file with the larger buffer-length parameter.

If access to a primary file uses sequential block buffering, access to all associated alternate-key records also uses it.

After FILE_SETKEY_ or KEYPOSITION with a nonzero key specifier, the first FILE_READ64_/READ request causes the disk process to fetch a data block from the alternate-key file into the buffer area. The disk process then fetches a single record from the primary data file

by using 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, but be aware that this combination can cause concurrency problems.

If another process is updating data copied into the block buffer, those updates might 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 disk. 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

You can have two or more files share the same buffer space by specifying identical sequential-block-buffer parameters in FILE_OPEN_. This can result in 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 an indication with warning 5 (failure to provide sequential buffering) 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 between files on successive reads, 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 Operating system, the file cannot share a buffer.

Specifying the Appropriate Disk File ACCESSTYPE Parameter

A parameter called ACCESSTYPE is associated with each disk file. This parameter in effect tells the disk process how to use its buffer and cache space when reading from and writing to the associated file. You use SETMODE 91 calls to examine or set ACCESSTYPE.

Choosing the most appropriate ACCESSTYPE for each of an application's files can increase the efficiency with which the disk process carries out its tasks. The various access modes are:

- Random access. When you specify random access, the disk process employs a least recently used (LRU) approach to reusing cache space. This practice tends to keep frequently used blocks in cache memory instead of reading them in from the disk every time they are needed.

- Sequential access. When you specify sequential access, the disk process uses essentially the same LRU algorithm that it does for random access but does so in such a way that the least recently used data blocks are removed from the cache sooner. With sequential access, the least recently used data block survives in cache approximately half as long as it would for random access. This practice prevents cache memory from being filled with data blocks that you probably do not need again.

- There are two cases that tend to call for this type of access: one being the truly sequential case and the other being the case where you are accessing a very large file randomly.

- System-managed access. When you specify system-managed access, the disk process decides whether you are accessing the file randomly or sequentially based upon your previous usage and optimizes its behavior accordingly. This is the default access type.

- Direct I/O. When you specify direct I/O access, the disk process bypasses cache memory completely if the file has been opened for write-through, unstructured access in either

exclusive read/write or protected read-only mode. This type of access might be desirable, for example, in an application that requires fast unstructured access and derives no real benefit from the use of cache memory.

Refreshing the End-of-File (EOF) Pointer

Each file's end-of-file (EOF) pointer is kept in its file control block (FCB) in main memory. To maximize performance, the EOF pointer is normally written to the file's disk label only when needed.

Although refreshing the file's disk label only under limited conditions maximizes system performance, certain considerations should be taken into account. These considerations do not apply in the case of TMF audited files, because the EOF is recovered by the autorollback feature.

- If an open file is backed up, the EOF pointer in the file label copy on tape does not reflect the actual state of the file. An attempt to restore such a file results in an error.

- If the system is shut down (each processor module has been reset) while a file is open, the EOF pointer in the file label on disk does 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 EOF pointer in the file label on disk will not reflect the actual state of the file.

There is an autorefresh option for Enscribe files that you can enable or disable by using FUP ALTER commands or FILE_CREATE_ system procedure calls. This option, when enabled, causes the disk label to be refreshed automatically each time the file label changes, including the EOF pointer field. This autorefreshing is always on for key-sequenced and queue files; any REFRESH setting is ignored.

The additional I/O caused by the REFRESH ON option can decrease processing throughput significantly. For applications that cannot afford this overhead, the EOF pointer in the file label on disk can be forced to represent the actual state of a file through periodic use of the DISK_REFRESH_ procedure. Execution of REFRESH writes the information contained in any FCBs to the file labels on the associated disk volume.

The REFRESH option can also be set with the PUP REFRESH command. The REFRESH command 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 the backup tape will represent the actual states of each file.

To use the equivalent SCF command, enter SCF CONTROL DISK $<volume>, REFRESH at the TACL prompt.

System operators use the REFRESH command before initiating a total system shutdown to ensure that all file labels on disk correctly represent the actual state of each file. The disk process, when idle, periodically refreshes changed EOF pointers to the file label on disk regardless of the various REFRESH option states.

Purging Data

Either the FUP PURGEDATA command or the CONTROL/FILE_CONTROL64_ 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 FILE_PURGE_ procedure or the FUP command PURGE can delete a file from the disk directory.

Following are four ways to logically purge data from an Enscribe file:

1. The FUP PURGE command
2. The FILE_PURGE_ system procedure
3. The FUP PURGEDATA command
4. The CONTROL 20 system procedure

None of these actually physically erases data. Instead, they all reset various pointers to indicate that either:

- The file no longer exists (FUP PURGE command or the FILE_PURGE_ system procedure)

- The file exists but contains no data (FUP PURGEDATA command or CONTROL 20 system procedure)

In the first case, if the file space is subsequently reallocated to another file, the new file's owner can read the logically purged data. For security reasons, you might want to avoid this. You can do so by enabling the CLEARONPURGE option prior to purging the file. You enable CLEARONPURGE by using either the SETMODE 1 system procedure or the FUP SECURE command. Having done so, all of the data in the file is physically erased (overwritten with zeros) when the file is purged. CLEARONPURGE has no effect in conjunction with the FUP PURGEDATA command.

For example, the TAL code resets the current-record, next-record, and EOF pointers of the particular file to point to relative byte 0 and updates the EOF pointer in the file label on disk:

```
LITERAL purgedata = 20;
CALL CONTROL ( filenum, purgedata );
IF < THEN...
```

The file still exists, but it now logically contains no data.

You can also use the CONTROL 21 system procedure (allocate/ deallocate extents) in conjunction with CONTROL 20 to logically purge a file's data and then deallocate all of its extents:

```
LITERAL purgedata = 20,
                alloc^op = 21,
                dealloc = 0;
CALL CONTROL ( filenum, purgedata );
IF < THEN ...
CALL CONTROL ( filenum, alloc^op, dealloc );
IF < THEN ...
```

The file still exists, but it now logically contains no data and all of its extents have been released (deallocated).

CONTROL 21 can be used to deallocate extent space beyond the EOF even when the EOF points to a relative byte greater than zero.

Programmatically Allocating File Extents

You can use the CONTROL 21 system procedure to allocate one or more file extents for an open file. For example, to allocate 16 extents to a newly created file, open the file and then issue a CONTROL 21 procedure call:

```
LITERAL alloc^op = 21,
                max^ext = 16;
CALL CONTROL ( filenum, alloc^op, max^ext );
IF < THEN ...
```

NOTE: If all extents cannot be allocated because the file label is full, error 43 is returned.

Programmatically Deallocating File Extents

You can also use the CONTROL 21 system procedure to deallocate any file extents beyond the extent to which the EOF pointer is currently pointing.

For example, to deallocate any unused extents in a file, open the file and then issue a CONTROL 21 procedure call:

```
LITERAL alloc^op = 21,
                dealloc  = 0;
CALL CONTROL ( filenum, alloc^op, dealloc );
```

The file still exists, but all file extents beyond the EOF extent are deallocated.

5 Unstructured Files

Enscribe Unstructured Files

An Enscribe unstructured file is essentially a byte array on disk that starts at byte address zero and continues sequentially upward through whatever byte address is identified by the end-of-file (EOF) pointer. The file system imposes no further structure on such files. How data is grouped into records and how records are ordered within the file are the responsibility of the application process.

The files created by the EDIT or TEDIT utilities, for example, are unstructured. What structure they do have is imposed by the utilities themselves, not by the Enscribe software. Such files can be read by the EDITREAD procedure, by the sequential I/O (SIO) routines, or by EDIT or TEDIT. EDIT and TEDIT files are identifiable by the file code 101.

Application designers typically use unstructured files for exclusive (nonshared), intermediate storage of fixed-length data records that are accessed sequentially.

Access to the data in unstructured files is accomplished using a relative byte address (RBA) maintained by the Enscribe software and the read-count or write-count parameter supplied by the application process in system procedure calls such as FILE_READ64_, FILE_WRITE64_, FILE_READUPDATE64_, READ[X], WRITE[X], and READUPDATE[X].

Applicable System Procedures

You use these system procedures to create and access Enscribe unstructured files:

- FILE_CREATE_, FILE_CREATELIST_

- FILE_OPEN_, FILE_CLOSE_, AWAITIO[X], FILE_AWAITIO64_

- FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, LOCKFILE, LOCKREC, UNLOCKFILE, UNLOCKREC

- FILE_SETPOSITION_, FILE_SAVEPOSITION_, FILE_RESTOREPOSITION_

- FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X], READUPDATELOCK[X]

- FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITE[X], WRITEUPDATE[X], WRITEUPDATEUNLOCK[X]

- FILE_GETINFO_, FILE_GETINFOLIST_, FILE_GETINFOBYNAME_, FILE_GETINFOLISTBYNAME_

- SETMODE, CONTROL, FILE_CONTROL64_

Types_Access

The types of access associated with unstructured files are sequential access, random access, and appending to the end of a file.

To perform sequential access, you use successive FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ[X], READLOCK[X], and WRITE[X] calls to operate upon consecutively higher blocks of data. To perform random access, you use the FILE_SETPOSITION_ system procedure to explicitly manipulate the content of the current-record and next-record pointers. You can also use FILE_SETPOSITION_ to change the content of the next-record pointer so that it points to the EOF position (for appending records to the end of the file); you do so by specifying the value -1 as the desired RBA.

Creating Unstructured Files

You create Enscribe unstructured files with the File Utility Program (FUP) or by calling either the FILE_CREATE_ procedure or the FILE_CREATELIST_ procedure. When you create an unstructured file, you must consider the buffer size and the disk extent sizes.

Buffer Size

The buffer size attribute lets you define the internal buffer size to be used by the disk process when accessing an unstructured file. The buffer size attribute can be any of these: 512 bytes, 1 KB, 2 KB, or 4 KB.

When creating a file by using the FILE_CREATE_ procedure, you use the blocklen parameter to specify the internal buffer size. When creating a file by using the FILE_CREATELIST_ procedure, you use item code 44 in the item-list to specify the internal buffer size.

When creating a file by using FUP, you use the BUFFERSIZE parameter to specify the desired buffer size.

When you open a file, the buffer size is automatically set to the value that was specified for the file when it was created.

During program execution, you can change the buffer size dynamically by specifying either a BUFFERSIZE value in a FUP ALTER command or by way of param1 in a SETMODE 93 procedure call

Note that the FUP ALTER command also changes the BUFFERSIZE parameter contained in the file label on disk. The SETMODE 93 call, however, applies only until you reverse it by using a FUP ALTER command or another SETMODE 93 system procedure call or until the file is closed.

If you specify an invalid buffer size, then the next higher valid size is used. The default buffer size is 4096, the maximum possible. Note that the buffer size you specify has no effect on the format of the data. Only 4 kb is supported for 512-b sector devices.

Because the buffer size attribute in effect defines the physical unit of transfer, the most efficient physical data transfers are those that start on a buffer-sized boundary and whose read count or write count is an integral multiple of the buffer size.

A buffer size that is exactly the same size as the anticipated transfer size lets the disk process use its fixed-length cache management scheme most efficiently. For example, if it is reasonable for your application to read and write its data in 1024-byte quantities and on 1024-byte boundaries (that is, the first block of data starts at RBA 0, the second at RBA 1023, the third at 2047, and so forth), the best buffer size attribute to specify would be 1024.

For example, suppose that you let the disk process use the default buffer size of 4096 and your application program naturally does its transfers in blocks of 1024 bytes. In such a case, each write of 1024 bytes will cause the disk process to first read a block of 4096 bytes, modify 1024 bytes of it in cache memory, and then write the 4096-byte block back to the disk. If the file is being audited by TMF or was opened with a syncdepth greater than zero, then a large audit-checkpoint (AC) record will also be generated because this appears to be a partial update to a 4096-byte record.

If, instead, you specified a buffer size of 1024 and you are always reading and writing in multiples of 1024, then each write of 1024 bytes will involve no read, a write of 1024 bytes, and no AC record at all.

The performance difference in these two cases can be dramatic. The best practice for unstructured files, whenever possible, is to always perform buffer-sized reads and writes starting at buffer-sized boundaries.

Disk Extent Size

When you create an Enscribe file you can specify the size of the primary and secondary extents (1 to 65,535 pages for a format 1 file, where a page is 2048 bytes). You can also specify the

maximum number of extents to be allocated for the file (16 or more for nonpartitioned unstructured files). If you do not specify extent sizes, both the primary and secondary extent sizes default to one page.

If you do not specify a maximum number of extents, MAXEXTENTS defaults to 16.

For nonpartitioned unstructured files, you can change the MAXEXTENTS value dynamically during program execution by using either the SETMODE 92 system procedure or the FUP ALTER command.

Restrictions on Partitioned Unstructured files

The following restrictions apply to partitioned unstructured files on an XP storage array and on H-series, J-series, and S-series internal disks:

- All extents must be a multiple of 14 pages.

- All partitions must have identical extent sizes and maximum extents, because partitioning depends on each partition having the same size.

Example: Creating an Unstructured File

The most efficient way to access unstructured disk files is by using buffer-sized reads and writes starting at buffer-sized boundaries.

The file created in this example is designed to store fixed-length logical records that are 512 bytes long. Provided that your application program will always read and write in multiples of 512 (thus maintaining alignment on buffer-sized boundaries), selecting a buffer size of 512 allows the disk process to perform most efficiently. If you designate the primary extent size as 6250 pages and the secondary extent size as 2500 pages, then the primary extent accommodates 25,000 of the 512-byte logical records and each secondary extent accommodates 10,000. When all 16 extents are eventually used, the entire file accommodates 175,000 records.

Assume also that you want to identify the file by the file code 1234. You could create the file by using the FUP commands:

```
> volume $volume1.subvol1
> fup
-set type u
-set code 1234
-set ext (6250,2500)
-set buffersize 512
-show
    TYPE U
    CODE 1234
    EXT ( 6250 PAGES, 2500 PAGES )
    BUFFERSIZE 512
    MAXEXTENTS 16
-create datafile
CREATED - $VOLUME1.SUBVOL1.DATAFILE
```

You could also use the FILE_CREATE_ procedure to create the file by including the TAL code in one of your application modules:

```
LITERAL name^length = 25,
        pri^extent = 6250,
        file^code = 1234,
        sec^extent = 2500,
        file^type = 0, ! file type = unstructured
        buffer^size = 512;
INT error;
INT filenum;
INT namelen;
STRING .filename [0:name^length-1] :=
                "$VOLUME1.SUBVOL1.DATAFILE";
namelen := name^length;
```

```
error := FILE_CREATE_ (filename:name^length,
        namelen, file^code, pri^extent,
        sec^extent, , file^type, , , buffer^size);
```

Accessing Unstructured Files

This subsection discusses the pointers and different types of access associated with unstructured files.

File Pointers

Three major pointers are associated with an Enscribe unstructured file:

Current-record pointer	Specifies the RBA of the location that was most recently read from or written to.
Next-record pointer	Specifies the RBA of the next location to be read from or written to.
EOF pointer	Specifies the RBA of the next even numbered byte following the last data byte in the file unless the odd unstructured option has been set for the file. If the odd unstructured option is set, the EOF occurs at the next sequential byte after the last data byte whether it is odd or even numbered.

When you open an unstructured file, both the current-record and next-record pointers point to the first byte in the file (RBA zero).

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

A single EOF pointer, however, is associated with all opens of a given unstructured disk file. This permits data to be appended to the end of a file by several different accessors. The EOF pointer resides in the file's file control block (FCB) in the disk I/O process environment. A file's EOF pointer value is copied from the file label on disk when the file is opened and is not already open. The system maintains a working copy of the file's EOF pointer in the FCBs that are in both the primary and backup system processes that control the associated disk volume.

You can explicitly change the content of the next-record pointer to that of the EOF pointer by specifying an address of -1 in a FILE_SETPOSITION_ call. When appending to a file, the EOF pointer is advanced automatically each time a new data record is added to the end of the file. Note that in the case of partitioned files, the EOF pointer relates only to the final partition containing data.

A file's EOF pointer is not automatically written through to the file label on disk each time it is modified. Instead, for unstructured files, it is physically written to the disk only when one of these events occurs:

- A file label field is changed and the autorefresh option is enabled.

- The last accessor of the file, closes the file.

- The DISK_REFRESH_ procedure is called for the file.

- The REFRESH command is executed for the file's volume.

When creating a file by using the FILE_CREATE_ system procedure, you enable or disable the autorefresh feature by setting bit 10 of the *options* parameter to 1 or 0, respectively.

When creating a file by using FUP, you enable or disable the autorefresh feature by specifying REFRESH or NO REFRESH, respectively, in the FUP CREATE command.

When you open a file, the autorefresh feature is enabled or disabled at that time depending upon what was specified for the file when it was created. You can enable or disable this feature

dynamically during program execution by specifying REFRESH or NO REFRESH, respectively, in a FUP ALTER command. When you do so, however, whatever you designate applies until you reverse it with a subsequent FUP ALTER command or until the file is closed.

Table 8 summarizes the values that the various pointers are set to upon conclusion of the particular system procedure.

In this table, *count* is the transfer count specified by the read or write procedure call. If the file is an odd unstructured file, the value specified by *count* is the number of bytes transferred. If the file is an even unstructured file, *count* is rounded up to an even number before the data is transferred.

Table 8 File-Pointer Action

System Procedure	Pointer Values
CONTROL (write EOF)	EOF pointer := next-record pointer; file label's EOF pointer := EOF pointer;
CONTROL (purge data)	current-record pointer := next-record pointer := EOF pointer := 0; file label's EOF pointer := EOF pointer;
CONTROL (allocate/ deallocate extents)	file pointers are unchanged; file label's EOF pointer := EOF pointer;
FILE_CLOSE_ (last)	file label's EOF pointer := EOF pointer;
FILE_CREATE_	file label's EOF pointer := 0;
FILE_CREATELIST_	file label's EOF pointer := 0;
FILE_OPEN_ (first)	EOF pointer := file label's EOF pointer;
FILE_OPEN_ (any)	current-record pointer := next-record pointer := 0;
FILE_SETPOSITION_	current-record pointer := next-record pointer := *relative-byte-address*;
READ[X] or READLOCK[X]	current-record pointer := next-record pointer; next-record pointer := next-record pointer +$min (*count*, EOF pointer - next-record pointer);
READUPDATE[X]	file pointers are unchanged
READUPDATELOCK[X]	file pointers are unchanged

WRITE[X]	`if next-record pointer = -1 then`
	`  begin`
	`    current-record pointer := EOF pointer;`
	`    EOF pointer := EOF pointer + count;`
	`  end`
	`else`
	`  begin`
	`    current-record pointer := next-record pointer;`
	`    next-record pointer := next-record pointer + count;`
	`    EOF pointer:= $max(EOF pointer, next-record pointer);`
	`  end;`

WRITEUPDATE[X]	file pointers are unchanged
WRITEUPDATEUNLOCK[X]	file pointers are unchanged

Legend

:= means "is set to"

$max represents a function in which the larger of the two specified values is used. $min represents a function in which the smaller of the two specified values is used.

Sequential Access

FILE_READ64_, FILE_READLOCK64_, READ[X], READLOCK[X], FILE_WRITE64_ and WRITE[X] operations increment the next-record pointer by the number of bytes transferred, thereby providing sequential access to the file.

If the file is an odd unstructured file, both the number of bytes transferred and the amount by which the pointers are incremented are exactly the number of bytes specified by the write count or read count parameter. If the file is an even unstructured file, the values of the write count and read count parameters 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 system procedure calls illustrates how the file pointers are used when sequentially accessing an unstructured disk file; the example assumes that these are the first operations after the file is opened:

```
CALL READ ( filenum, buffer, 512 );
CALL READ ( filenum, buffer, 512 );
CALL WRITEUPDATE ( filenum, buffer, 512 );
CALL READ ( filenum, buffer, 512 );
```

The first READ transfers 512 bytes into the designated buffer starting at relative byte 0. Upon completion of this READ operation, the next-record pointer points to relative byte 512 and the current-record pointer points to relative byte 0.

The second READ transfers 512 bytes into the buffer starting at relative byte 512. Upon completion of this READ operation, the next-record pointer 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 disk location. The file system transfers 512 bytes from the 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 altered by the WRITEUPDATE operation.

The third READ transfers 512 bytes into the buffer starting at relative byte 1024 (the address in the next-record pointer). Upon completion of this READ operation, the nextrecord pointer points to relative byte 1536 and the current-record pointer points to relative byte 1024.

Encountering the EOF During Sequential Reading.

If you encounter the EOF boundary while reading an unstructured disk file, the data up to the EOF location is transferred. A subsequent read request will return an EOF indication (condition code CCG) because it is not permissible to read past the EOF location. If you do not alter the pointers by using a FILE_SETPOSITION_ call, the EOF indication will be returned for every subsequent read request.

For example, consider an unstructured file with the EOF location at relative byte 4096, as illustrated in Figure 12 (page 65). Assume that an application program executes this sequence of 512-byte reads starting at relative byte 0:

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

Figure 12 Example of Encountering the EOF
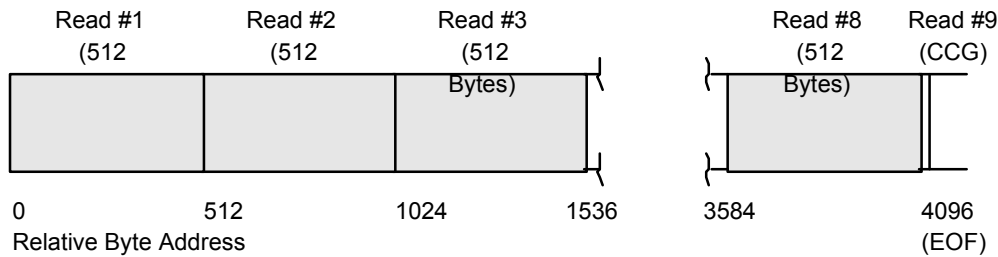
Each of the first eight READ calls transfers 512 bytes into the designated buffer, returns a num^read of 512, and sets the condition code to CCE (operation successful).

The ninth READ fails, no data is transferred into the buffer, num^read is returned as zero, and the condition code CCG indicates that you encountered the EOF.

If the read count is changed from 512 to 400, the results of executing the same read loop are somewhat different, as illustrated in

In this case, the first 10 READ calls each transfer 400 bytes into the designated buffer, return a num^read of 400, and set the condition code to CCE (operation successful). The eleventh READ transfers 96 bytes into the buffer, returns a num^read of 96, and sets the condition code to CCE. The twelfth READ fails and sets the condition code to CCG.

Figure 13 Example of Encountering the EOF (Short READ)

Random Access
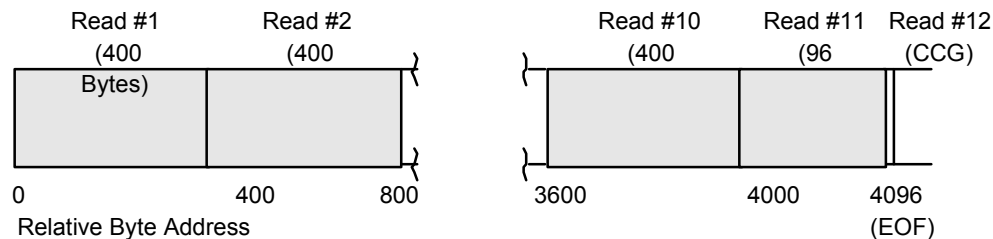
You access Enscribe unstructured disk files by using the relative-byte-address parameter of the POSITION system procedure to explicitly set the file pointers. To update data in an unstructured disk file at relative byte address 81920, you could use this sequence:

```
CALL FILE_SETPOSITION_ ( filenum, 81920F );
CALL READUPDATE ( filenum, buffer, 512 );
CALL WRITEUPDATE ( filenum, buffer, 512 );
```

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

The call to READUPDATE transfers 512 bytes from the disk to the designated buffer, starting at relative byte 81920 in the disk file. Following the read operation, the file pointers are unchanged.

The WRITEUPDATE procedure replaces the just-read data with new data in the same location on disk. The file system transfers 512 bytes from the buffer to the file at relative byte 81920. Following the write operation, the file pointers are unchanged.

Appending to the End of a File

You can also use the FILE_SETPOSITION_ procedure to append data to the end of an unstructured disk file. To set the next-record pointer to the EOF position, pass -1 as the relative byte address parameter in a FILE_SETPOSITION_ call:

```
CALL FILE_SETPOSITION_ ( filenum, -1F );
```

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

This WRITE call, if issued immediately after the FILE_SETPOSITION_ -1 call, appends 512 bytes to the end of the file:

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

The file system transfers 512 bytes from the designated buffer to the relative byte address pointed to by the EOF pointer. Upon completion of the WRITE operation, the EOF pointer is incremented by 512, the current-record pointer points to the old EOF position, and the next-record pointer still contains -1. A subsequent WRITE will also append to the end of the file.

6 Key-Sequenced Files

Enscribe Key-Sequenced Files

> **NOTE:** As of the H06.28 and J06.17 RVUs, format 2 legacy key-sequenced 2 (LKS2) files with increased limits, format 2 standard queue files with increased limits, and enhanced key-sequenced (EKS) files with increased limits are introduced. EKS files with increased limits support 17 to 128 partitions along with larger record, block, and key sizes. LKS2 files with increased limits and format 2 standard queue files with increased limits support larger record, block, and key sizes. When a distinction is not required between these file types, key-sequenced files with increased limits is used as a collective term. To use increased Enscribe limits, the minimum RVUs are H06.28 and J06.17 with specific SPRs. (These SPR requirements could change or be eliminated with subsequent RVUs.) For a list of the required H06.28/J06.17 SPRs, see SPR Requirements for Increased Enscribe Limits for the H06.28/J06.17 Release (page 18).

> **NOTE:** As of the H06.22 and J06.11 RVUs, a new Enscribe file type: enhanced key-sequenced (EKS) file is introduced and extends the Enscribe partition maximum to 64. An EKS file supports 17 to 64 partitions. The previous Enscribe key-sequenced file type supports 1 to 16 partitions and is referred to as a legacy key-sequenced (LKS) file in this manual. When a distinction is not required between these file types, key-sequenced file is used as a collective term.

Enscribe key-sequenced files consist of variable-length records that are accessed by the values contained within designated key fields. There are two types of keys: primary and alternate. All records in a key-sequenced file contain a primary key. The use of alternate keys is optional.

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. The records in a key-sequenced file are stored logically in ascending order according to the value contained in their primary-key field.

A record can vary in length from one byte to the maximum record size specified when the file was created. For the maximum record sizes of key-sequenced files, see Table 7 (page 46). 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 inserted (with the restriction that the length cannot exceed the specified maximum record size). Records in a key-sequenced file can be deleted.

A good example of the use of key-sequenced files in an application environment is an inventory file in which each record describes a part. The primary key field for that file would probably be the part number, so the file would be ordered by part number. Other fields in the record would contain such information as vendor name, quantity on hand, and so forth, and one or more of them can be designated as alternate key fields. Note that the Enscribe software is concerned only with key fields; the content and location of all other fields in each record is solely the concern of the application.

Applicable System Procedures

Use these system procedures to create and access Enscribe key-sequenced files:

- FILE_CREATE_, FILE_CREATELIST_

- FILE_OPEN_, FILE_CLOSE_, AWAITIO[X], FILE_AWAITIO64_

- FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, LOCKFILE, LOCKREC, UNLOCKFILE, UNLOCKREC

- FILE_SETKEY_, FILE_SETKEYPOSITION_, FILE_SAVEPOSITION_, FILE_RESTOREPOSITION, KEYPOSITION, SAVEPOSITION, REPOSITION

- FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_,
  FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X],
  READUPDATELOCK[X]

- FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_,
  WRITE[X], WRITEUPDATE[X], WRITEUPDATEUNLOCK[X]

- FILE_GETINFO_, FILE_GETINFOLIST_, FILE_GETINFOBYNAME_,
  FILE_GETINFOLISTBYNAME_

- SETMODE, CONTROL, FILE_CONTROL64_

Types of Access

Key-sequenced files can be accessed either sequentially or randomly. Sequential access is preferable, for example, when generating a report of the quantity on hand of all parts in an inventory file. Random access is preferable when you want to identify the vendor of a particular part.

When you read from a key-sequenced file by primary key, each FILE_READ64_/READ operation retrieves the record containing the next sequentially higher primary-key value. Similarly, when you read by an alternate key, each FILE_READ64_/READ operation retrieves the record containing the next sequentially higher value in the specified alternate-key field.

The FILE_SETKEY_ or KEYPOSITION system procedure specifies which access path (primary key or a particular alternate-key field) you want to use and the field value at which you want to start.

If you do not use FILE_SETKEY_ or KEYPOSITION, access is by primary key and begins with the first record in the file.

You can use FILE_SETKEY_ or KEYPOSITION during program execution to dynamically change the access path and the current-record pointer.

Key-Sequenced Tree Structure

Key-sequenced files are physically organized as one or more bit-map blocks and a B-tree structure of index blocks and data blocks. Figure 14 (page 69) illustrates a sample key-sequenced file tree structure. Bit-map blocks within the file organize the free space of a structured file.

Each data block contains a header plus one or more data records, depending on the record size and data-block size. 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 disk 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 in a similar manner: 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 disk process must generate a new level of indexes. The disk process 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 disk process must do this again each time the highest-level block is split.

The disk 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.

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, disk space usage is less efficient, and performance is degraded.

Typically, in a changing database most blocks will be approximately two-thirds full at any given time. When using the FUP LOAD command to load data into a key-sequenced file, you can specify how much empty space to provide for future growth.

Figure 14 Key-Sequenced B-Tree Structure
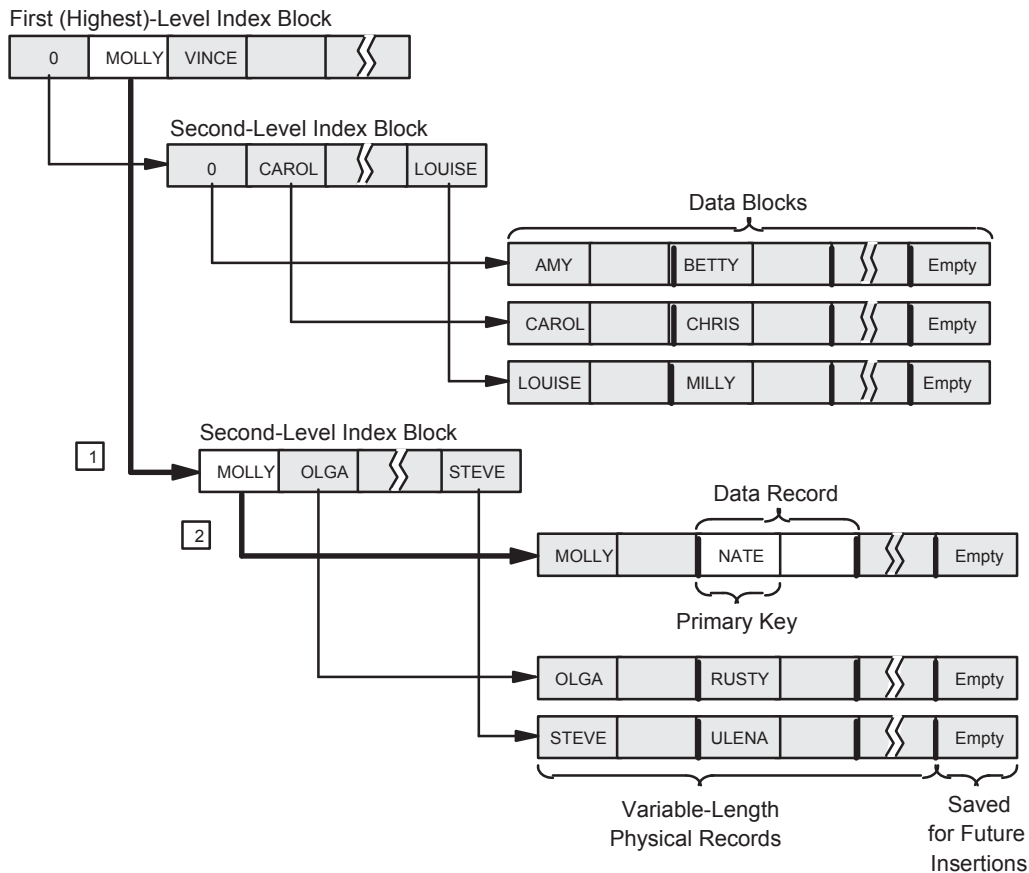
Legend

1 NATE is alphabetically greater than MOLLY but less than VINCE.
   Go to the second-level index block that begins with MOLLY.

2 NATE is alphabetically greater than MOLLY but less than OLGA.
   Go to the data block that begins with MOLLY.

Note that data records are never chained together in Enscribe key-sequenced files. Instead, the tree structure is dynamically rebalanced to ensure that all records in the file can be accessed with the same number of FILE_READ64_/READ operations, that number being the number of levels of indexes plus one for the data block.

Unique Features of EKS Files

Enhanced Key-Sequenced Files (EKS) have these unique features:

- The primary partition does not store user data but rather stores file meta data

- When creating an EKS file, the partial key of the first secondary partition must be explicitly set to all zeros.

- When creating an EKS file, the primary extent size of the primary partition is adjusted based on the block length of the file. The secondary extent size for the primary partition is ignored.

- When accessing the primary partition of an EKS file in unstructured mode, reads on the file return FEEOF (1) and writes return FEFILEFULL (45). This is because the meta data stored in the primary partition cannot be accessed by non-privileged applications.

Creating Key-Sequenced Files

You create Enscribe legacy, legacy with increased limits, enhanced key-sequenced files, or enhanced key-sequenced files with increased limits with the File Utility Program (FUP) or by calling either the FILE_CREATE_ procedure or the FILE_CREATELIST_ procedure.

If you are using FUP to create a file that contains one or more alternate-key fields, FUP automatically creates any required alternate-key files. If you are creating the file programmatically, however, you must create any required alternate-key files yourself (one for each alternate-key field).

When creating a key-sequenced file, your considerations include:

Comparing LKS Files, LKS2 Files, Standard Queue Files, EKS Files, and EKS Files with Increased Limits

Table 9 Comparison of LKS Files, LKS2 Files, Standard Queue Files, EKS Files, and EKS Files with Increased Limits

File Attribute	LKS and Standard Queue File Attributes Allowed Values	LKS2 and Standard Queue File Attributes Allowed Values	EKS File Attributes Allowed Values	EKS Files with Increased Limits Allowed Values
Format Type	1 or 2	2	2	2
Number of Partitions	Up to 16	Up to 16	17 up to 64[1]	17 up to 128
Extents for Primary Partition	Up to 978	Up to 928	Up to 928 [2]	Up to 928
Extents for Secondary Partition	Up to 978	Up to 928	Up to 928	Up to 928
Partition File Extent Sizes	Each partition has its own primary and secondary extent size. The maximum extent size for a format 1 file is 65535 pages, while the maximum extent size for a format 2 file is 536,870,912 pages.	Each partition has its own primary and secondary extent size. The maximum extent size for a format 2 file is 536,870,912 pages.	The primary partition has a primary extent size greater than or equal to 140 pages and cannot be less than 140 pages.[3] Each secondary partition has its own primary and secondary extent size.	The primary extent size of the primary partition is based on the block length of the file. The secondary extent size for the primary partition is ignored. Each secondary partition has its own primary

Table 9 Comparison of LKS Files, LKS2 Files, Standard Queue Files, EKS Files, and EKS Files with Increased Limits *(continued)*

File Attribute	LKS and Standard Queue File Attributes Allowed Values	LKS2 and Standard Queue File Attributes Allowed Values	EKS File Attributes Allowed Values	EKS Files with Increased Limits Allowed Values
				and secondary extent size.
Block Length	512, 1 KB, 2 KB, 4 KB	512, 1 KB, 2 KB, 4 KB (With increased limits, 32 KB is supported.)	512, 1 KB, 2 KB, 4 KB	512, 1 KB, 2 KB, 4 KB, 32 KB
Record Length	Format 1: Up to 4062 Format 2: Up to 4040	Up to 4040 (With increased limits, up to 27,648.)	Up to 4040	Up to 27,648
Primary Key Length	Up to 255	Up to 255 (With increased limits, up to 2048.)	Up to 255	Up to 2048
Alternate Key Length	Up to ~255 (Not applicable to Enscribe Queue files.)	Up to ~255 (With increased limits, up to ~2048.) (Not applicable to Enscribe Queue files.)	Up to ~255	Up to ~2048
Partition Partial Key Length	Up to 255[4] (Not applicable to Enscribe Queue files.)	Up to 255[4] (Not applicable to Enscribe Queue files.)	Up to 255	Up to 255
Number of Alternate Key Files	Up to 100[4] (Not applicable to Enscribe Queue files.)	Up to 100[4] (Not applicable to Enscribe Queue files.)	Up to 100	Up to 100
Maximum Partition Size	Format 1: ~2 GB – 1 MB Format 2: ~1 TB	~1 TB	~1 TB	~1 TB
Maximum File Size	Format 1: ~32 GB – 16 MB Format 2: ~16 TB	~16 TB	~63 TB	~127 TB

[1] The primary partition of an enhanced key-sequenced file cannot contain user data and is used instead to store a portion of the file's label.

[2] Although the maximum number of extents for the primary partition of an EKS files is configurable, in practice DP2 will never allocate more than 1 extent.

[3] Hewlett Packard Enterprise reserves the right to change this value in the future. The actual extent size will be increased to the minimum if necessary.

[4] Or limited by the file label space.

ⓘ **IMPORTANT:**   You might not be able to create key-sequenced files with all possible combinations of number of secondary partitions, partial key length, and number of alternate keys because of file label space limitations. For example, if you attempt to create a key-sequenced file with 15 secondary partitions, 255-byte partial keys, and 100 alternate key files, the file creation API returns an error 1027, indicating that the file cannot be created because of label restrictions. To resolve this problem, reduce the number of secondary partitions, partial key length, or the number of alternate keys.

Converting a Legacy Key-Sequenced File to an Enhanced Key-Sequenced File

The following scenario is an example where the number of partitions for a customer database is increased beyond 16 partitions with a reconfiguration of the number of partitions, the number of extents per partition, and the partition key for the first secondary partition.

A customer application revolves around the use of an Enscribe key-sequenced file with 16 partitions. Over time the number of requestors for the application has increased and I/O on the key-sequenced file has become a bottleneck. The customer decides to increase the number of partitions from 16 to 64 in order to provide higher disk I/O throughput and to also increase the maximum size of the database from approximately 16 GB to approximately 200 GB. The customer brings down their application, does a backup of the legacy key-sequenced file, then uses FUP to create a new enhanced key-sequenced file with all attributes like the legacy file except the following:

Table 10 Converting LKS Files to EKS Files

File Attribute	Legacy Key-Sequenced File	Enhanced Key-Sequenced File
Format Type	2	2
Number of Partitions	16	64
Maximum Number of Extents for the Primary Partition File	928	16*
Maximum Number of Extents for the Secondary Partition File	928	928
Primary Extent Size of Primary Partition (in Pages)	536	28
Secondary Extent Size of Primary Partition (in Pages)	536	28
Primary Extent Size of all Secondary Partitions (in Pages)	536	1702
Secondary Extent Size of all Secondary Partitions (in Pages)	536	1702
Partition Key of First Secondary Partition (Partition Key Length of 15)**	"BAAAAAAAAAAAAAA"	[0,0,0,0,0,0,0,0,0,0,0 ,0,0,0,0]

*The primary partition of an enhanced key-sequenced file does not contain user data and is fixed at 16 maxextents with a primary extent size of 28 pages. Note that internally only the primary extent of the primary partition is allocated and used to store the file's partition array information.

**The partition partial key for the first secondary partition is always a fixed value of all zeroes. In a legacy key-sequenced file, the primary partition has an implicit partition partial key value of all zeroes and the first secondary partition has a partial key value greater than all zeroes

Logical Records

A logical record is the unit of information transferred between an application program and the file system.

When creating a key-sequenced file you must specify the maximum logical record length of that file. The particular maximum record size that you choose when creating a file depends upon the particular requirements of your application.

Key-Sequenced Format 1 Files

A logical record can be up to the block-size minus 34 bytes long.

Key-Sequenced Format 2 Files

A logical record can be up to the block-size minus 56 bytes long.

Using the maximum allowed block size of 4096, the absolute maximum logical record size allowed for a legacy key-sequenced file is 4062 bytes. Key-sequenced files with increased limits do not have this 4062 byte limitation. Key-sequenced files with increased limits have a 27,648 byte limit.

The data records that you write to a key-sequenced file can be of varying lengths, but none can exceed the maximum logical record size specified when the file was created. If you try to write a record that is longer than the defined maximum record length, the file system rejects the operation and returns an error 21 (illegal count).

Blocks

A block is the unit of information transferred between the disk process and the disk (or, when you are using sequential block buffering, between the disk process and the process file segment). A block consists of one or more logical records and, in the case of key-sequenced files, associated control information. This control information, which is used only by the system, is summarized in "Block Formats of Structured Files" (page 172).

The block size of a key-sequenced file with increased limits can be 512 bytes, 1 KB, 2 KB, 4 KB, or 32 KB.

Regardless of the record length, the maximum number of records that can be stored in a single block is 511 for a format 1 file. For a format 2 file, it is limited by the block and record sizes.

A record cannot span block boundaries (that is, it cannot begin in one block and end in another). Therefore, the block size for a key-sequenced file must be at least:

- *record-length* + 2 + 32 bytes for format 1 files

- *record-length* + 4 + 52 bytes for format 2 files

For key-sequenced files, the disk process requires that the size of the file's index blocks (IBLOCK parameter of a FUP SET command) and data blocks (BLOCK parameter of a FUP SET command) be the same. Accordingly, when you are creating a key-sequenced file, the disk process ignores whatever IBLOCK size you might specify and uses instead (as both the BLOCK and IBLOCK values) whatever you specify for BLOCK.

When choosing the block size, remember that while longer index blocks require more space in the cache buffer they can also reduce the number of indexing levels (thereby reducing the number of accesses to the disk).

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

Disk Extent Size

When you create an Enscribe file, you can specify these:

- The size of the primary and secondary extents. The primary partition for format 2 files have a primary extent size greater than or equal to 140 pages. Each secondary partition has its own primary and secondary extent size. The maximum extent size for a format 2 file is 536,870,912 pages. Format 1 files are restricted to: 1 to 65,535 pages, where a page is 2048 bytes.

- The maximum number of extents to be allocated for the file (16 or more for a key-sequenced file or any of its partitions).

- The system defaults to format 2 if any of the following is true:

  ○ Secondary partition is greater than 15

  ○ Primary or alternate key size is greater than 255

- ○ Block size is greater than 4 KB

- ○ Record size is greater than 4 KB

If you do not specify extent sizes, both the primary and secondary extent sizes default to one page.

If you do not specify the maximum number of extents, MAXEXTENTS defaults to 16.

For key-sequenced files and any of their partitions, you can change the MAXEXTENTS value dynamically during program execution using either the SETMODE 92 system procedure or the FUP ALTER command.

Primary Keys

For key-sequenced files, you must define both the offset from the beginning of the record where the primary-key field begins and the length of the key field. A few things to consider when choosing the offset of the primary-key field are:

- The primary-key field can begin at any offset within a record and can be of any length up to 2048:

  $min(*record-length* minus *offset*, 2048)

- If you will be using key compression in data blocks, the primary-key field must reside at the very beginning of the record.

- If the primary-key field is the final field in the record, it can be of variable length.

- If the key field is to be treated as a data type other than STRING, the *offset* should be chosen so that the field begins on a word boundary.

Note that the collating sequence by which the records within a key-sequenced file are arranged is by ASCII code (actually unsigned binary). Consequently, if the data type of the key field is binary, the presence of the sign bit will cause negative values to be treated as being greater than positive values.

Key Compression

When creating a file, you can specify that the keys be compressed in data and/or index blocks.

The software compresses keys by eliminating leading characters that are duplicated from one key to the next and replacing them with a 1-byte count of the duplicate characters. For example, if these three records are inserted into a file with data compression enabled:

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

what is actually written to the disk is:

```
0JONES, JANE
8OHN
7SAM
```

where the first character (0, 8, and 7, respectively) indicates the number of leading characters that are identical to those of the primary key in the immediately preceding record.

When you are creating a file by using FUP, the DCOMPRESS, ICOMPRESS, and COMPRESS parameters of the SET command designate whether key compression is to be applied to the data blocks, the index blocks, or both. When you are creating a file by using the FILE_CREATE_ procedure, bits 11 and 12 of the options parameter designate what type of key compression, if any, is to be used. If you use the FILE_CREATELIST_ procedure, item codes 68 and 69 designate key compression.

When deciding whether or not to use key compression, consider these:

- Key compression can require one additional byte per record. Moreover, key compression will always require additional system processing to expand the compressed records.
- Key compression requires that the primary-key field begin at offset [0] of each record. Consequently, you cannot use variable-length primary keys unless the entire record is the primary-key field.
- If there is considerable similarity among the records' primary-key values, then key compression in data blocks is desirable.
- If there is enough similarity among records that the first records of successive blocks have similar primary-key values, then key compression of index blocks is also desirable.
- Key compression in data blocks is useful for alternate-key files where several alternate keys tend to have the same value.

Index Compaction

A separate mechanism is also automatically used to make all index records more compact regardless of whether key compression is in effect. Index compaction differs from key compression in that it eliminates the trailing portion of similar records, whereas key compression eliminates the leading portion.

In an index block of a key-sequenced file there is one index record for each block (data or index) below it in the tree. This index record is formed by comparing the first primary key of the block with the last primary key of the previous block. If the two 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 the four data blocks:

```
ALLEN, HARRY      FRASER, IAN      JONES, JOHN      LARIMER, JO
ARKIN, ALAN       GAULT, WILLY     KILMER,JOYCE     LORE, KEVIN
     :                 :                :                :
EICHER, DAVE      HAM, JACK        LAINE, LOIS      MAILER, NORM
FRANKLIN, BEN     JONES,JANE       LANSON,SAM       MARNER, SID
```

These three index records are actually written to disk:

```
FRAS
JONES, JO
LAR
```

File Creation Examples

The pages that follow present annotated examples showing how to create:

1. A key-sequenced file
2. A key-sequenced file with alternate keys
3. A key-sequenced format 2 file with alternate keys
4. An alternate-key file
5. A partitioned key-sequenced file

Example 1: Creating a Key-Sequenced File

This example shows how to create a credit file in which the individual records are to be accessed by customer name.

The record format is:

Byte Offset:

0	34	134	142	150

name	address	current-balance	limit

Primary-Key
Field

With a record size of 150, selecting a block size of 2048 results in a blocking factor of 13 data records per block:

N = (block-size - 32) / (record-size + 2)

13 = (2048 - 32) / (150 + 2)

If you designate the primary extent size as 5000 pages and the secondary extent size as 2000 pages, then the primary extent will accommodate 65,000 credit records and each secondary extent will accommodate 26,000 additional credit records. When all 16 extents are eventually used, the file will accommodate a total of 455,000 credit records.

The primary-key length is 34 bytes.

Assume also that you want to identify the file by the file code 1000 and that you want to enable key compression for both data and index blocks. You could create the file by using these FUP commands:

```
> volume $store1.svol1
> fup
-set type k
-set code 1000
-set ext (5000,2000)
-set rec 150
-set block 2048
-set compress
-set keylen 34
-show
    TYPE K
    CODE 1000
      EXT ( 5000 PAGES, 2000 PAGES )
      REC 150
      BLOCK 2048
      IBLOCK 2048
      KEYLEN 34
      KEYOFF 0
      DCOMPRESS, ICOMPRESS
-create myfile
CREATED - $STORE1.SVOL1.MYFILE
```

Using the FILE_CREATE_ system procedure, you could create the file by including this TAL code in one of your application modules.

In this code, the volume name and node name are not specified. FILE_CREATE_ obtains them from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE. For more information on the =_DEFAULTS DEFINE, see the *TACL Programming Guide*.

```
LITERAL name^length = 12,
        pri^extent = 5000,
        file^code = 1000,
        sec^extent = 2000,
        file^type = 3,
        options = %30, !data compression
        rec^len = 150,
        data^block^len = 2048,
        key^len = 34,
```

```
            key^offset = 0;
INT error;
INT namelen;
STRING .filename [0:name^length-1] :=
      "SVOL1.MYFILE";


namelen := name^length;
error := FILE_CREATE_ (filename:name^length, namelen,
          file^code, pri^extent, sec^extent,, file^type,
          options, rec^len, data^block^len, key^len,
          key^offset);
IF Error <> 0 THEN ... ! error
```
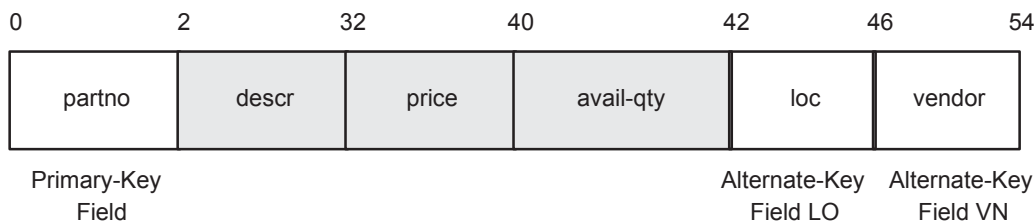
Example 2: Creating a Key-Sequenced File With Alternate Keys

This example shows how to create a key-sequenced inventory control file in which the primary
key is the part number and both the storage location code and vendor number are alternate keys.

You could create the file by using these FUP commands:

```
volume $store1.svol1
fup
-set type k
-set code 1001
-set ext (32,8)
-set rec 54
-set block 4096
-set 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 4096
    KEYLEN 2
    KEYOFF 0
    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
```

Using the FILE_CREATELIST_ system procedure, you could create the file by including this TAL
code in one of your application modules.

The volume name, subvolume name, and node name are not specified in the procedure call.
FILE_CREATELIST_ obtains them from the current value of the VOLUME attribute of the
=_DEFAULTS DEFINE.

```
      ?NOLIST
      ?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
      ?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
      ?                                 FILE_OPEN_,
      ?                                 FILE_CREATELIST_,
      ?                                 READ)
      ?LIST

      PROC DO^THE^WORK MAIN;
      BEGIN
      LITERAL name^length = 3,
              num^altkeys = 2,
              num^altkey^files = 1,
              item^list^len = 13;
      INT error;
      INT error2;
      INT namelen;
      STRING .filename [0:name^length-1] := "INV";

      INT .item^list [0:item^list^len-1];

      STRUCT value^list;
        BEGIN
          INT file^type;
          INT file^code;
          INT rec^len;
          INT block^len;
          INT key^offset;
          INT key^length;
          INT pri^extent;
          INT sec^extent;
          INT altkeys;
          STRUCT altkey^descr [0:num^altkeys-1];
            BEGIN
              STRING key^specifier [0:1];
              INT key^length;
              INT key^offset;
              INT key^filenum;
              INT null^value;
              INT attributes;
            END;
          INT num^alt^key^files;
          STRUCT name^length^info [0:num^altkey^files-1];
            BEGIN
              INT file^name^len;
            END;
          STRING file^names [0:5];
          END;

      namelen := name^length;

      item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                      ZSYS^VAL^FCREAT^FILECODE,
                      ZSYS^VAL^FCREAT^LOGICALRECLEN,
                      ZSYS^VAL^FCREAT^BLOCKLEN,
                      ZSYS^VAL^FCREAT^KEYOFFSET,
                      ZSYS^VAL^FCREAT^KEYLEN,
                      ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                      ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                      ZSYS^VAL^FCREAT^NUMALTKEYS,
                      ZSYS^VAL^FCREAT^ALTKEYDESC,
                      ZSYS^VAL^FCREAT^NUMALTKEYFILES,
                      ZSYS^VAL^FCREAT^ALTFILELEN,
                      ZSYS^VAL^FCREAT^ALTFILENAMES ];
      value^list.file^type := 3;       ! key-sequenced
```

```
value^list.file^code := 1001;
value^list.rec^len := 54;
value^list.block^len := 4096;
value^list.key^offset := 0;
value^list.key^length := 2;
value^list.pri^extent := 32;
value^list.sec^extent := 8;
value^list.altkeys := num^altkeys;
value^list.altkey^descr[0].key^specifier ':=' "LO";
value^list.altkey^descr[0].key^length := 4;
value^list.altkey^descr[0].key^offset := 42;
value^list.altkey^descr[0].key^filenum := 0;
value^list.altkey^descr[0].null^value := 0;
value^list.altkey^descr[0].attributes := 0;
value^list.altkey^descr[1].key^specifier ':=' "VN";
value^list.altkey^descr[1].key^length := 8;
value^list.altkey^descr[1].key^offset := 46;
value^list.altkey^descr[1].key^filenum := 0;
value^list.altkey^descr[1].null^value := 0;
value^list.altkey^descr[1].attributes := 0;
value^list.num^alt^key^files := num^altkey^files;
value^list.name^length^info[0].file^name^len := 6;
value^list.file^names ':=' "INVALT";

ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
        item^list, item^list^len, value^list,
        $LEN(value^list), error2);
END;
```

When you use a system procedure to create your key-sequenced file, you must create your alternate key files separately. For more information, see Accessing Key- Sequenced Files"Accessing Key-Sequenced Files" (page 85)

Example 3: Creating a Key-Sequenced Format 2 File With Alternate Keys

This example creates the same file as in "Example 2: Creating a Key-Sequenced File With Alternate Keys" (page 77), except that it creates a format 2 file.

You could create the file by using these FUP commands:

```
volume $store1.svol1
fup
-set type k
-set code 1001
-set ext (32,8)
-set format 2
-set rec 54
-set block 4096
-set keylen 2
-set altkey ("LO",keyoff 42,keylen 4)
-set altkey ("VN",keyoff 46,keylen 8)
-set altfile (0,invalt)
-show
    TYPE K
    FORMAT 2
    CODE 1001
    EXT ( 32 PAGES, 8 PAGES )
    FORMAT 2
    REC 54
    BLOCK 4096
    IBLOCK 4096
    KEYLEN 2
    KEYOFF 0
    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
```

Using the FILE_CREATELIST_ system procedure, you could create the file by including this TAL code in one of your application modules.

The volume name, subvolume name, and node name are not specified in the procedure call. FILE_CREATELIST_ obtains them from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE.

```
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                                 FILE_OPEN_,
?                                 FILE_CREATELIST_,
?                                 READ)
?LIST

PROC DO^THE^WORK MAIN;
BEGIN

LITERAL name^length = 3,
        num^altkeys = 2,
        num^altkey^files = 1,
        item^list^len = 14;
INT error;
INT error2;
INT namelen;
STRING .filename [0:name^length-1] := "INV";

INT .item^list [0:item^list^len-1];

STRUCT value^list;
  BEGIN
    INT file^type;
    INT file^code;
    INT rec^len;
    INT block^len;
    INT key^offset;
    INT key^length;
    INT pri^extent;
    INT sec^extent;
    INT fileformat; !format 2
    INT altkeys;
    STRUCT altkey^descr [0:num^altkeys-1];
      BEGIN
        STRING key^specifier [0:1];
        INT key^length;
        INT key^offset;
        INT key^filenum;
        INT null^value;
        INT attributes;
      END;
    INT num^alt^key^files;
    STRUCT name^length^info [0:num^altkey^files-1];
      BEGIN
        INT file^name^len;
      END;
STRING file^names [0:5];
END;

namelen := name^length;

item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
```

```
                          ZSYS^VAL^FCREAT^FILECODE,
                          ZSYS^VAL^FCREAT^LOGICALRECLEN,
                          ZSYS^VAL^FCREAT^BLOCKLEN,
                          ZSYS^VAL^FCREAT^KEYOFFSET,
                          ZSYS^VAL^FCREAT^KEYLEN,
                          ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                          ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                          ZSYS^VAL^FCREAT^FILEFORMAT, !format 2
                          ZSYS^VAL^FCREAT^NUMALTKEYS,
                          ZSYS^VAL^FCREAT^ALTKEYDESC,
                          ZSYS^VAL^FCREAT^NUMALTKEYFILES,
                          ZSYS^VAL^FCREAT^ALTFILELEN,
                          ZSYS^VAL^FCREAT^ALTFILENAMES ];
value^list.file^type := 3; ! key-sequenced
value^list.file^code := 1001;
value^list.rec^len := 54;
value^list.block^len := 4096;
value^list.key^offset := 0;
value^list.key^length := 2;
value^list.pri^extent := 32;
value^list.sec^extent := 8;
value^list.fileformat := 2; !format 2
value^list.altkeys := num^altkeys;
value^list.altkey^descr[0].key^specifier ':=' "LO";
value^list.altkey^descr[0].key^length := 4;
value^list.altkey^descr[0].key^offset := 42;
value^list.altkey^descr[0].key^filenum := 0;
value^list.altkey^descr[0].null^value := 0;
value^list.altkey^descr[0].attributes := 0;
value^list.altkey^descr[1].key^specifier ':=' "VN";
value^list.altkey^descr[1].key^length := 8;
value^list.altkey^descr[1].key^offset := 46;
value^list.altkey^descr[1].key^filenum := 0;
value^list.altkey^descr[1].null^value := 0;
value^list.altkey^descr[1].attributes := 0;
value^list.num^alt^key^files := num^altkey^files;
value^list.name^length^info[0].file^name^len := 6;
value^list.file^names ':=' "INVALT";
ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
          item^list, item^list^len, value^list,
          $LEN(value^list), error2);

END;
```

When you use a system procedure to create your key-sequenced file, you must create your alternate key files separately. For more information, see "Accessing Key-Sequenced Files" (page 85).

Example 4: Creating an Alternate-Key File

When you use FUP to create the primary file, FUP automatically creates any required alternate-key files. If you create the primary file programmatically, however, you must create the alternate-key file yourself as a separate operation.

You could create the alternate-key file for "Example 2: Creating a Key-Sequenced File With Alternate Keys" (page 77) by including this TAL code in one of your application modules.

The volume name, subvolume name, and node name are not specified in the procedure call. FILE_CREATE_ obtains them from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE.

```
LITERAL name^length = 6,
           pri^extent = 32,
           file^code = 1002,
           sec^extent = 8,
           file^type = 3,
           rec^len = 12,
```

```
                    data^block^len = 4096,
                    options = %10, ! data compression
                    key^len = 12, ! max. alternate key length
                                   ! + primary-key length
                                   ! + 2
                    key^offset = 0;
INT error;
INT namelen;
STRING .filename [0:name^length-1] := "INVALT";

namelen := name^length;

error := FILE_CREATE_(filename:name^length, namelen,
        file^code, pri^extent, sec^extent,, file^type,
        options, rec^len, data^block^len, key^len,
        key^offset);
IF Error <> 0 THEN ... ! error
```

Example 5: Creating a Partitioned, Key-Sequenced File

This example shows how to create a key-sequenced file that will span six partitions. The record format is the same as in "Example 1: Creating a Key-Sequenced File" (page 75)

Byte Offset:

0	34	134	142	150

name	address	current-balance	limit

Primary-Key
Field

The file is to reside on six volumes and be partitioned in this manner:

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	MINNEAPOLIS, MN	NO	0000.00	0500.00
ROGERS	BOISE, ID	WE	1024.00	1500.00
SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

NOTE: If the file is an enhanced key-sequenced file (with more than 16 partitions), then the primary partition is reserved for internal configuration information and does not contain data. In this case, you must set the partition key value for the first secondary partition to zero.

To create the file with FUP, include SET PART commands to describe the partitioning.

```
> volume $part0.svol1
> fup
-set type k
-set code 1000
-set ext (50,20)
-set rec 150
-set block 4096
-set keylen 34
-set part (1,$part1,50,20,"DA")
-set part (2,$part2,50,20,"HA")
-set part (3,$part3,50,20,"LA")
-set part (4,$part4,50,20,"PA")
-set part (5,$part5,50,20,"TA")
-show
    TYPE K
    CODE 1000
      EXT ( 50 PAGES, 20 PAGES )
      PART ( 1, $PART1, 50 PAGES, 20 PAGES, "DA")
      PART ( 2, $PART2, 50 PAGES, 20 PAGES, "HA")
      PART ( 3, $PART3, 50 PAGES, 20 PAGES, "LA")
      PART ( 4, $PART4, 50 PAGES, 20 PAGES, "PA")
      PART ( 5, $PART5, 50 PAGES, 20 PAGES, "TA")
      REC 150
      BLOCK 2048
      IBLOCK 2048
      KEYLEN 34
      KEYOFF 0
-create custfile
CREATED - $PART0.SVOL1.CUSTFILE
```

To create a partitioned file when using the FILE_CREATELIST_ system procedure, place the partition information into the value array. Place the appropriate value codes into the item list.

This TAL code creates a key-sequenced file partitioned across six volumes:

```
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                               FILE_OPEN_,
?                               FILE_CREATELIST_,
?                               READ)
?LIST

PROC DO^THE^WORK MAIN;
BEGIN

LITERAL name^length = 26,
        num^partitions = 5,
        item^list^len = 14;

INT error;
INT error2;
INT namelen;

STRING .filename[0:name^length-1]:=
      "\SYS.$PART0.SVOL1.CUSTFILE";

INT .item^list [0:item^list^len-1];
STRUCT value^list;
```

```
   BEGIN
     INT file^type;
     INT file^code;
     INT rec^len;
     INT block^len;
     INT key^offset;
     INT key^length;
     INT pri^extent;
     INT sec^extent;
     INT partitions;
     STRUCT part^info [0:num^partitions-1];
       BEGIN
         INT part^pri^extent;
         INT part^sec^extent;
       END;
     STRUCT vol^name^len [0:num^partitions-1];
       BEGIN
         INT vol^name^act^len;
       END;
     STRING vol^names [0:29];
     INT part^part^key^len;
     STRING part^part^key^val [0:9];
   END;

   namelen := name^length;
   item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                   ZSYS^VAL^FCREAT^FILECODE,
                   ZSYS^VAL^FCREAT^LOGICALRECLEN,
                   ZSYS^VAL^FCREAT^BLOCKLEN,
                   ZSYS^VAL^FCREAT^KEYOFFSET,
                   ZSYS^VAL^FCREAT^KEYLEN,
                   ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                   ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                   ZSYS^VAL^FCREAT^NUMPRTNS,
                   ZSYS^VAL^FCREAT^PRTNDESC,
                   ZSYS^VAL^FCREAT^PRTNVOLLEN,
                   ZSYS^VAL^FCREAT^PRTNVOLNAMES,
                   ZSYS^VAL^FCREAT^PRTNPARTKEYLEN,
                   ZSYS^VAL^FCREAT^PRTNPARTKEYVAL ];
   value^list.file^type := 3;
   value^list.file^code := 1000;
   value^list.rec^len := 150;
   value^list.block^len := 4096;
   value^list.key^offset := 0;
   value^list.key^length := 34;
   value^list.pri^extent := 50;
   value^list.sec^extent := 20;
   value^list.partitions := 5;
   value^list.part^info[0].part^pri^extent := 50;
   value^list.part^info[0].part^sec^extent := 20;
   value^list.part^info[1].part^pri^extent := 50;
   value^list.part^info[1].part^sec^extent := 20;
   value^list.part^info[2].part^pri^extent := 50;
   value^list.part^info[2].part^sec^extent := 20;
   value^list.part^info[3].part^pri^extent := 50;
   value^list.part^info[3].part^sec^extent := 20;
   value^list.part^info[4].part^pri^extent := 50;
   value^list.part^info[4].part^sec^extent := 20;
   value^list.vol^name^len.vol^name^act^len[0] := 6;
   value^list.vol^name^len.vol^name^act^len[1] := 6;
   value^list.vol^name^len.vol^name^act^len[2] := 6;
   value^list.vol^name^len.vol^name^act^len[3] := 6;
   value^list.vol^name^len.vol^name^act^len[4] := 6;
   value^list.vol^names ':=' "$PART1$PART2$PART3$PART4$PART5";
   value^list.part^part^key^len := 2;
```

```
value^list.part^part^key^val ':=' "DAHALAPATA";

ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
        item^list, item^list^len, value^list,
        $LEN(value^list), error2);
END;
```

Byte counts for the values array are listed in the *System Procedure Calls Reference Manual*.

Accessing Key-Sequenced Files

The following paragraphs discuss the end-of-file pointer and how to access Enscribe key-sequenced files.

End-of-File (EOF) Pointer

An EOF pointer is associated with each disk file and is shared by all opens of that file. For key-sequenced files this pointer contains the relative byte address of the byte following the last (highest address) block that currently contains data. Note that a key-sequenced file can have empty blocks interspersed among in-use blocks. When you are adding data to a key-sequenced file, the EOF pointer increments each time a new block is added because there are no empty blocks.

The system maintains a working copy of the file's EOF pointer in the file control blocks (FCBs) that are in both the primary and backup system processes that control the associated disk volume. For Enscribe key-sequenced files, the EOF pointer is physically written to the disk when any of these events occurs: any file label field is changed, the last accessor closes the file, the DISK_REFRESH_ system procedure is called for the file or the REFRESH command is executed for the file's volume.

Sequential Access

You perform sequential processing, in which a related subset of records is read in ascending order within the current access path, by using the FILE_READ64_, FILE_READLOCK64_, READ and READLOCK system 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 EOF indication.

The first call to FILE_READ64_, FILE_READLOCK64_, READ or READLOCK after a file-opening or positioning operation reads the record (if any) at the current position. Subsequent calls to FILE_READ64_, FILE_READLOCK64_, READ or READLOCK, without intermediate positioning, return successive records (if any) in the designated subset.

After each call to FILE_READ64_, FILE_READLOCK64_, READ or READLOCK, the position of the returned record becomes the current position.

Random Access

You perform random access processing by using the FILE_READUPDATE64_, FILE_WRITEUPDATE64_, FILE_READUPDATELOCK64_, FILE_WRITEUPDATEUNLOCK64_, READUPDATE, WRITEUPDATE, READUPDATELOCK, and WRITEUPDATEUNLOCK system procedures. The update 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), the file system returns error code 11 (record not found).

You cannot use FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE or WRITEUPDATEUNLOCK to alter a record's primary key. If you need to do so, you must first

delete the record and then reinsert it (using a FILE_WRITE64_/WRITE call) with the new key value.

If updating or locking is attempted immediately after a call to FILE_SETKEY_ or KEYPOSITION where a non-unique alternate key is specified, the updating or locking fails with an error 46 (invalid key). However, if an intermediate call to FILE_READ64_, FILE_READLOCK64_, READ or READLOCK is performed, then the updating or locking is permitted.

Inserting Records

You perform record insertion by using the FILE_WRITE64_/WRITE system procedure. Insertion requires 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 fails with an error 10 (record already exists). If the operation is part of a TMF transaction, the record is locked for the duration of the transaction.

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 fails with an error 10 (record already exists).

Insertion of an empty record (where the write count parameter of the FILE_WRITE64_/WRITE call is zero) is not valid for key-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 it is not, the insertion fails with an error 21 (invalid count).

Deleting Records

You perform record deletion by using the FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE or WRITEUPDATEUNLOCK system procedure with a write count of zero. Record deletion always applies to the current position in a file.

Current Primary-Key Value

A key-sequenced file's current primary-key value is taken from the primary key associated with the last FILE_READ64_, FILE_READLOCK64_, READ or READLOCK operation or FILE_SETKEY_ or KEYPOSITION operation by primary key.

After opening a key-sequenced file, but before issuing the first read or FILE_SETKEY_ or KEYPOSITION call, the current primary-key value is that of the first record in the file.

Access Examples

The access examples throughout the remainder of this section all use the customer record definition:

Byte Offset:

0	34	134	136	144	152
name	address	region	curbal	limit

Primary-Key
Field

Alternate-Key
Field RG

The TAL declaration of a customer record is:

```
LITERAL name^len = 34,
         address^len = 100,
         region^len = 2;
```

```
STRUCT customer^record (*);
  BEGIN
    STRING cust^name [0:name^len - 1];       ! name
    STRING cust^address [0:address^len - 1]; ! address
    STRING cust^region [0:region^len - 1];   ! region
                                             ! NO = northern
                                             ! SO = southern
                                             ! EA = eastern
                                             ! WE = western
    FIXED (2) cust^curbal; ! current balance
    FIXED (2) cust^limit; ! credit limit
  END;
STRUCT .cust^rec (customer^record);
```

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	MINNEAPOLIS, MN	NO	0000.00	0500.00
ROGERS	BOISE, ID	WE	1024.00	1500.00
SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

Example 1: Action of Current Position

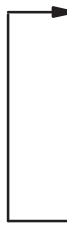This sample TAL code shows how to position by the primary-key value ROGERS:

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

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

Successive read calls will then access the unshaded records in this illustration, in the order shown:

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	MINNEAPOLIS, MN	NO	0000.00	0500.00
ROGERS	BOISE, ID	WE	1024.00	1500.00
SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

Current Position After
KEYPOSITION Call

This sample TAL code shows how to position by the alternate-key value NO within the RG access path:

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

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

Successive read calls will then access the unshaded records in this illustration, in the order shown:

Current Position After
KEYPOSITION Call

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	MINNEAPOLIS, 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, A	REEDLEY, CA	WE	0256.95	0300.00
EVANS	BUTTE, MT	WE	0010.00	0100.00
ROGERS	BOISE, ID	WE	1024.00	1500.00
SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00

Example 2: Approximate Subset by Primary Key After OPEN

```
INT error;
INT .cust^filename[0:11],
    cust^filenum;

error := FILE_OPEN_(cust^filename:name^length, cust^filenum);
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop
    CALL READ (cust^filenum, cust^rec, $LEN(cust^rec) );
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error
      ELSE
        BEGIN ! process the record
          ...
        END;
  END;      ! read loop
```

Because no positioning is done between the OPEN call and the read loop, access is by the primary-key value and starts with the first record in the file. There are 11 data records in the sample file. The read loop is executed 12 times. The first 11 read calls return data records and set the condition code to CCE, indicating a successful completion. The twelfth read call returns no data and sets the condition code to CCG, indicating that the EOF was encountered.

This illustration visually represents the results of each read call executed within the read loop:

	Primary-Key Field				
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	RICHARDS	MINNEAPOLIS, MN	NO	0000.00	0500.00
9	ROGERS	BOISE, ID	WE	1024.00	1500.00
10	SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00
11	SMITH	DAYTON, OH	NO	0010.00	0500.00
12	EOF				

Example 3: Approximate Subset by Alternate Key

This sample TAL code shows how to perform approximate positioning by the alternatekey value EA within the RG access path:

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

! position to the first record that contains EA in the
!   region field
key ':=' "EA";
CALL KEYPOSITION ( cust^filenum, key, "RG" );
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop
    CALL READ (cust^filenum, cust^rec, $LEN(cust^rec) );
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error
      ELSE
        BEGIN ! process the record
         ...
        END;
  END;    ! read loop
```

The KEYPOSITION call sets the current position at the first record in the primary data file that contains the value EA in the region (RG) field. Access is by the alternate-key access path RG.

Successive read calls within the read loop access all of the records in the primary data file that contain the value EA or greater in the RG field. When two or more records contain the same value in the RG field, those records are accessed in ascending order by the primary key. In addition to returning a data record, each read call sets the condition code to CCE, indicating successful completion.

The final read call in the read loop returns no data and sets the condition code to CCG, indicating that the EOF was encountered. This illustration visually represents the results of each read call executed within the read loop.

			Alternate- Key Field 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	MINNEAPOLIS, 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, A	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	LOS ANGELES, CA	WE	0301.00	1000.00
12	EOF				

Example 4: Generic Subset by Primary Key

This sample TAL code shows how to perform generic positioning by the primary-key value BROWN:

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

key ':=' "BROWN";
compare^len := 5;
generic = 1;
```

```
CALL KEYPOSITION ( cust^filenum , key , ,
                      compare^len , generic );

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

The KEYPOSITION call sets the current position at the first record in the primary data file that contains the value BROWN as the first five characters in the primary-key field. Access is by the primary-key access path.

Successive read calls within the read loop access all of the records in the primary data file that contain the value BROWN as the first five characters in the primary-key field. Within that generic subset, records are accessed in ascending order by the primary key. In addition to returning a data record, each read call sets the condition code to CCE, indicating successful completion.

The final read call in the read loop returns no data and sets the condition code to CCG, indicating that the EOF was encountered. This illustration visually represents the results of each read call executed within the read loop:

Primary-Key
Field

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

This sample TAL code shows how to perform exact positioning by the primary-key value SMITH:

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

key ':=' "SMITH";
exact = 2;
CALL KEYPOSITION ( cust^filenum, key, , , exact );

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

```
        END;
  END;       ! read loop
```

The KEYPOSITION call sets the current position at the first record in the primary data file that contains the value SMITH as the first five characters in the primary-key field. Access is by the primary-key access path.

Exact positioning by a primary-key value always gives you access to only one record in the primary data file: the record at the current position after the KEYPOSITION call.

The read loop is executed twice. The first time through the loop, the read call returns the specified data record and sets the condition code to CCE, indicating successful completion. The second time through the loop, the read call returns no data and sets the condition code to CCG, indicating that the EOF was encountered. This illustration visually represents the results of each read call executed within the read loop:

Primary-Key
Field

1	SMITH	DAYTON, OH	NO	0010.00	0500.00

2   EOF

Example 6: Generic Subset by Nonentity Alternate Key

This sample TAL code shows how to perform generic positioning by the alternate-key value NO within the RG access path:

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

key ':=' "NO";
generic = 1;
CALL KEYPOSITION ( cust^filenum, key, "RG",, generic );

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

The KEYPOSITION call sets the current position at the first record in the primary data file that contains the value NO as the first two characters in the region (RG) field. Access is by the alternate-key access path RG.

Successive read calls within the read loop access all of the records in the primary data file that contain the value NO as the first two characters in the RG field. Within that generic subset, records are accessed in ascending order by the primary key. In addition to returning a data record, each read call sets the condition code to CCE, indicating successful completion.

The final read call in the read loop returns no data and sets the condition code to CCG, indicating that the EOF was encountered. This illustration visually represents the results of each read call executed within the read loop:

1	HARTLEY	CHICAGO, IL	NO	0433.29	0500.00
2	RICHARDS	MINNEAPOLIS, 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

This sample TAL code shows how to insert a new record into a key-sequenced primary data file:

```
! blank the customer record

cust^rec ':=' " " & cust^rec FOR $LEN(cust^rec) -1 BYTES;

! move the new data into cust^rec

cust^rec.cust^name    ':=' "HEATHCLIFF";
cust^rec.cust^address ':=' "PORTLAND, OR.";
cust^rec.cust^region  ':=' "WE";
cust^rec.cust^curbal  :=   0.00F;
cust^rec.cust^limit   :=   500.00F;

! write the new record to disk

CALL WRITE (cust^filenum, cust^rec, $LEN(cust^rec) );
IF <> THEN ... ! error
```

The contents of the *customer* file after the insertion 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
HEATHCLIFF	PORTLAND, OR	WE	0000.00	0500.00
JONES	DALLAS, TX	SO	1234.56	2000.00
KOTTER	NEW YORK, NY	EA	0089.00	0500.00
RICHARDS	MINNEAPOLIS, MN	NO	0000.00	0500.00
ROGERS	BOISE, ID	WE	1024.00	1500.00
SANFORD	LOS ANGELES, CA	WE	0301.00	1000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

Inserted

Example 8: Random Update

This example shows how to select a single data record and then change the content of one of the fields in that record.

The KEYPOSITION call sets the current position at the record that contains the value HARTLEY in the primary-key field. The READUPDATE call reads the data record into the application buffer without altering the file pointers.

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

key ':=' "HARTLEY";
CALL KEYPOSITION (cust^filenum, key);
IF <> THEN ...
CALL READUPDATE (cust^filenum, cust^rec, $LEN(cust^rec) );
IF <> THEN ...
```

The data record read from the disk file into the application buffer is:

HARTLEY	CHICAGO, IL	NO	0433.29	0500.00

This sample TAL code changes the value in the current balance field of the record in the application buffer and then writes the updated record from the buffer to the disk without altering the file pointers.

```
    :
  cust^rec.cust^curbal := cust^rec.cust^curbal + 30.00F
    :
```

```
CALL WRITEUPDATE (cust^filenum, cust^rec, $LEN(cust^rec) );
IF <> THEN ...
```

The data record written from the application buffer to the disk file is:

HARTLEY	CHICAGO, IL	NO	0463.29	0500.00

Example 9: Random Update of a Nonexistent Record

This example shows an attempt to update a nonexistent record. Because the KEYPOSITION procedure does no searching of indexes, the attempt to access the nonexistent record is not discovered until the subsequent READUPDATE call.

The first four records in the primary data file are:

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

This sample TAL code tries to set the current position at the data record that contains the value BROWN,C in the primary-key field, and then tries to read that record into the application buffer in preparation for an update operation:

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

key ':=' "BROWN,C";
exact := 2;
CALL KEYPOSITION (cust^filenum, key, , , exact );
IF <> THEN ...
CALL READUPDATE (cust^filenum, cust^rec, $LEN(cust^rec) );
IF < THEN

  BEGIN
    status := FILE_GETINFO_ (cust^filenum, error);
    IF error = 11 THEN ... ! record not found
      :
  END;
```

The attempt to read the nonexistent record fails with a condition code of CCL and a file-system error code 11.

Example 10: Sequential Reading via Primary Key With Updating

This sample TAL code changes the content of the limit field to 2000.00 within any data record whose limit field value is currently >= 1000.00 and < 2000.00:

```
! position to the first record by primary key
compare^len := 0;
CALL KEYPOSITION ( cust^filenum, key, , compare^len);

cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop
```

```
         CALL READ ( cust^filenum, cust^rec, $LEN(cust^rec) );
         IF > THEN cust^eof := 1 ! end-of-file
         ELSE
           IF < THEN ... ! error
           ELSE
             BEGIN ! process the record
               IF cust^rec.cust^limit >= 1000.00F
               AND cust^rec.cust^limit < 2000.00F THEN
                 BEGIN
                   cust^rec.cust^limit := 2000.00F;
                   CALL WRITEUPDATE
                     (cust^filenum, cust^rec, $LEN(cust^rec) );
                   IF < THEN ... ! error
                 END;
             END;
      END;    ! read loop
```

This illustration shows the contents of the *customer* file after all of the applicable records have been updated:

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
EVANS	BUTTE, MT	WE	0010.00	0100.00
HARTLEY	CHICAGO, IL	NO	0463.29	0500.00
HEATHCLIFF	PORTLAND, OR	WE	0000.00	0500.00
JONES	DALLAS, TX	SO	1234.56	2000.00
KOTTER	NEW YORK, NY	EA	0089.00	0500.00
RICHARDS	MINNEAPOLIS, MN	NO	0000.00	0500.00
ROGERS	BOISE, ID	WE	1024.00	2000.00
SANFORD	LOS ANGELES, CA	WE	0301.00	2000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

Changed Fields

Example 11: Deleting a Record

This example shows how to select and delete the data record whose primary-key value is EVANS.

This illustration shows the contents of the applicable portion of the *customer* file before the deletion:

BROWN, B	BOSTON, MA	EA	0301.00	2000.00
EVANS	BUTTE, MT	WE	0010.00	0100.00
HARTLEY	CHICAGO, IL	NO	0433.29	0500.00

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

key ':=' "EVANS";
CALL KEYPOSITION (cust^filenum, key);
CALL WRITEUPDATE (cust^filenum, cust^rec , 0);
IF <> THEN ... ! error
```

This illustration shows the contents of the applicable portion of the customer file after the deletion:

BROWN, B	BOSTON, MA	EA	0301.00	2000.00
HARTLEY	CHICAGO, IL	NO	0433.29	0500.00

Example 12: Sequential Reading With Deletions Using Primary Key

This sample TAL code reads the customer file sequentially by primary-key value and deletes any records whose current balance field contains 0.00:

```
! position to the first record by primary key
compare^len := 0;
CALL KEYPOSITION ( cust^filenum, key, , compare^len);

cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop
    CALL READ ( cust^filenum, cust^rec, $LEN(cust^rec) );
    IF > THEN cust^eof := 1 ! end-of-file
    ELSE
      IF < THEN ... ! error
      ELSE
        BEGIN ! process the record
          IF cust^rec.cust^curbal = 0.00F THEN
            BEGIN
              CALL WRITEUPDATE ( cust^filenum, cust^rec, 0 );
              IF < THEN ... ! error
               :
            END;
        END;
  END;      ! read loop
```

This illustration shows the contents of the customer file after all of the applicable records have been deleted:

BROWN, A	REEDLEY, CA	WE	0256.95	0300.00
BROWN, B	BOSTON, MA	EA	0301.00	2000.00
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
ROGERS	BOISE, ID	WE	1024.00	2000.00
SANFORD	LOS ANGELES, CA	WE	0301.00	2000.00
SMITH	DAYTON, OH	NO	0010.00	0500.00

Example 13: Relational Processing

This example illustrates relational processing in which fields from records in one file are used to access data records in other files.

The example uses four files:

- The same *customer* file used in Examples 1 through 12

- An *order* file

- An *order detail* file

- An *inventory* file

The format of an order record is:
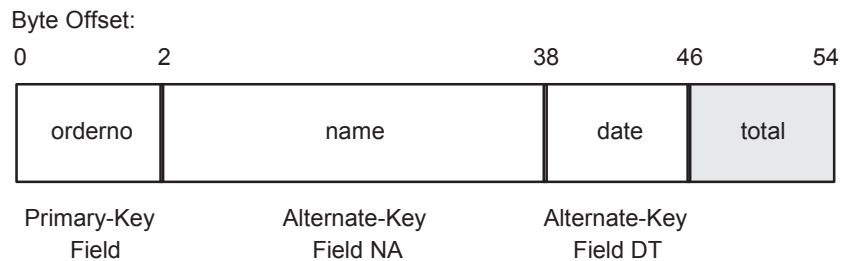
The TAL definition of an order record is:

```
LITERAL name^len = 36,
date^len = 8;

STRUCT order^struct (*);                   ! order record
  BEGIN
    FIXED(2) order^orderno;                ! order number
    STRING order^name [0:name^len - 1];    ! name
    STRING order^date [0:date^len - 1];    ! date
    FIXED(2) order^total;                  ! total = 0 means
  END;                                     ! order not filled;
                                           ! total <> 0 means
                                           ! order filled but
                                           ! not shipped
STRUCT .order (order^struct);
```

The contents of the order file are:

orderno	name	date	total
0020	SMITH	95/09/30	0000.00
0021	JONES	95/10/01	0000.00
0176	BROWN, B	95/10/17	0000.00
0410	SANFORD	95/10/22	0000.00
0498	ROGERS	95/11/02	0000.00
0601	SMITH	95/11/08	0000.00
0622	HARTLEY	95/11/12	0000.00
0623	KOTTER	95/11/12	0000.00

The format of an order detail record is:

Byte Offset:

0	2	4	6	8	16

orderno	itemno	partno	qty	itemtot

Primary-Key Field   Alternate-Key Field PN
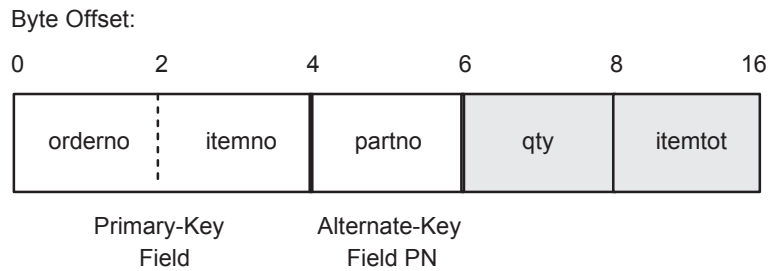
The TAL definition of an order detail record is:

```
STRUCT order^detail^struct (*); ! order detail record
  BEGIN
    FIXED(2) orderdet^orderno;              ! order number
    FIXED(2) orderdet^itemno;               ! item number
    FIXED(2) orderdet^partno;               ! part number
    FIXED(2) orderdet^itemtot;              ! total=0 means item
  END;                                      ! not available
STRUCT .orderdet (order^detail^struct);
```

The contents of the order detail file are:

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
0410	0001	01234	00010	0000.00
0410	0002	03379	00010	0000.00

. . .

| 0623 | 0012 | 01234 | 00010 | 0000.00 |

The format of an inventory record is:

Byte Offset:

0	2	32	40	42	46	54

partno	descr	price	availqty	loc	vendor

| Primary-Key Field | | | Alternate-Key Field AQ | Alternate-Key Field LO | Primary-Key Field VN |
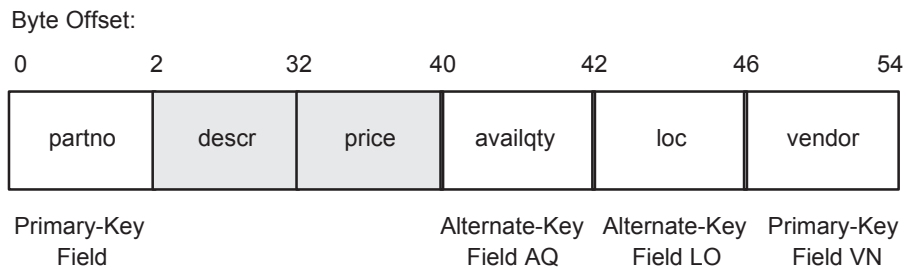
The TAL definition of an inventory record is:

```
LITERAL descr^len = 30,
        loc^len = 4,
        vendor^len = 8;

STRUCT inventory^struct (*);                 ! inventory record
  BEGIN
    FIXED(2) inv^partno;                     ! part number
    STRING inv^descr [0:descr^len - 1];  ! description
    FIXED(2) inv^price; ! price
    FIXED(2) inv^availqty;                   ! available quantity
    STRING inv^loc [0:loc^len - 1];      ! location
    STRING inv^vendor [0:vendor^len - 1];! vendor
  END;

STRUCT .inv (inventory^struct);
```

The contents of the inventory file are:

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 sample TAL code shown later in this example finds all of the orders that are more than one month old and then fills those orders; the code does so in five steps:

1. Using the date alternate-key field, read a record from the order file. The read loop terminates upon encountering a record whose date field contains a value greater than or equal to a specified order^limit date.

2. Using the name from the order record, read the appropriate customer record from the customer file. Using information from both the order record and the customer record, print an order header consisting of the order number and the customer name and address.

3. Using the order number from the order record, read the associated generic subset from the order detail file.

4. Each record in the order detail file represents one line item. For each line item in the appropriate generic subset, use the part number from the order detail record to read and update the appropriate inventory record in the inventory file; then update the order detail record and print the line item.

5. After all of the line items for the current order have been processed, update the total field of the order record to reflect the total price of the order. Using the name from the order record, update the current balance field in the appropriate customer record. Print the order total.

The sample TAL code is:

```
! position to beginning of file via date field
compare^len := 0;
CALL KEYPOSITION (order^filenum, key, "DT", compare^len);
order^eof := 0;
WHILE NOT order^eof DO
  BEGIN ! reading order file via date field
```

```
CALL READ (order^filenum, order, $LEN(order) );
IF > OR order.order^date >= limit^date THEN order^eof :=1
ELSE
  BEGIN ! fill order
    ! read customer file
    CALL KEYPOSITION (cust^filenum, order.order^name);
    CALL READUPDATE (cust^filenum,cust^rec,
                              $LEN(cust^rec) );
    ! print the order header
    ! read order detail file for current order
    compare^len := 2;
    generic := 1;
    CALL KEYPOSITION (orderdet^filenum,
       order.order^orderno, , compare^len, generic);
    orderdet^eof := 0;
    WHILE NOT orderdet^eof DO
      BEGIN
        ! read line item
         CALL READ
              (orderdet^filenum, orderdet,
                          $LEN(orderdet) );
         IF > THEN orderdet^eof := 1
         ELSE
          BEGIN
            CALL KEYPOSITION (inv^filenum,
                    orderdet.orderdet^partno);
            CALL READUPDATE (inv^filenum, inv,
                    $LEN(inv) );
             : ! update the inventory record
            CALL WRITEUPDATE (inv^filenum, inv,
                    $LEN(inv) );
             : ! update the order detail record
            CALL WRITEUPDATE (orderdet^filenum,
                    orderdet, $LEN(orderdet) );
            ! print the line item
          END;
        END;
      : ! update the order record
     CALL WRITEUPDATE (order^filenum, order,
                    $LEN(order) );
     : ! update the customer record
     CALL WRITEUPDATE (cust^filenum, cust^rec,
                              $LEN(cust^rec) );
     : ! print the total
  END; ! of fill order
END; ! reading order file via date field
```

The records used for filling the first order are:

From the *order* file:

0020	SMITH	95/09/30	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

The following are the contents of those same records after filling the first order. In the `order` file:

0020	SMITH	95/09/30	0147.25

In the `customer` file:

SMITH	DAYTON, OH	NO	0157.25	0500.00

In the `order detail` file:

0020	0001	23167	00000	0045.00
0020	0002	02010	00001	0000.00
0020	0003	12950	00000	0102.25

Line Item not Filled (Put on Back-Order)

In the *inventory* file:

23167	ANTENNA	0022.50	00006	A01	TAYLOR
02010	TOASTER	0022.50	00000	F22	ACME
12950	TOASTER	0020.45	00005	C98	SMYTHE

No Changes

7 Queue Files

Enscribe Queue Files

An Enscribe queue file is a special type of key-sequenced disk file that can function as a queue. Processes can queue and dequeue records in a queue file.

Queue files contain variable-length records that are accessed by values in designated key fields. Unlike other key-sequenced files, queue files have primary keys but cannot have alternate keys. The primary key for a queue file includes an 8–byte timestamp; you can add a user key if desired. The disk processes inserts the timestamp when each record in inserted into the file and maintains the timestamp during subsequent file operations.

Queue files provide these features:

- Access by multiple requester or queuing processes, with multiple servers or dequeuing processes allowed. Queue files are typically shared between multiple write processes and one read process, and are typically used for fairly low volume work.

△ **CAUTION:** All waiting readers are dispatched for every transaction or queue read. Use of queue files with multiple sub-queues can cause high utilization of CPU resources by the disk process and significantly affect performance.

- Protection against data loss with TMF. TMF is the main functional component of the TM/MP product.

- Flexible record ordering and selection; records can be prioritized; classed, or grouped as needed by an application; the default ordering is approximately first-in first-out (using the timestamp as the primary key).

- Record-level locking to prevent incomplete information from being accessed and to ensure that only one reader dequeues a specific record.

- Notification when new records are added to the file.

These restrictions apply to queue files:

- Queue files cannot be SQL objects.

- You cannot define alternate keys or partitions for queue files.

The first part of this section describes queue file structure and discusses how to access queue files. The remainder of the section contains examples showing how to create, open, and access queue files.

NOTE: Enscribe queue files should not be confused with Queue Manager queue files as described in the *Queue Manager Manual*. Despite the similarity in their names, they are entirely different types of files.

Applicable System Procedures

Use these system procedures to create and access Enscribe queue files:

- FILE_CREATE_, FILE_CREATELIST_, CREATE

- FILE_OPEN_, FILE_CLOSE_, AWAITIO[X], FILE_AWAITIO64_

- FILE_SETKEY_, FILE_SAVEPOSITION_, FILE_RESTOREPOSITION_, KEYPOSITION, SAVEPOSITION, REPOSITION

- FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X], READUPDATELOCK[X]

- FILE_WRITE64_, WRITE[X]
- FILE_GETINFO_, FILE_GETINFOLIST_, FILE_GETINFOBYNAME_, FILE_GETINFOLISTBYNAME_

Types of Access

Like key-sequenced files, queue files can be accessed by applications either sequentially or randomly.

Queue File Structure

Queue files are physically organized as one or more bit-map blocks and a B-tree structure of index and data blocks. Organization is the same as for key-sequenced files, described in "Key-Sequenced Files" (page 67). Queue files are distinguished from other key-sequenced files by having item 48 of the FILE_GETINFOLIST_procedure set to 1. The superseded FILEINFO procedure has bit 9 set in the *file-type* word if the file is a queue file.

Primary Keys

Each record in a queue file consists of a primary key and data. At a minimum, the primary key consists of an 8–byte timestamp generated by the disk process when a record is inserted in the file. In addition, you can define a user key that precedes the timestamp within the primary key.

The disk process maintains the timestamp as part of each record. This ensures that each record has a unique key and it eliminates the need to use an application-defined key for insertion or deletion of records. When you read a record from a queue file, the timestamp is part of the returned data. When you write a record, the disk process places the timestamp in the low-order eight bytes of the key, overwriting any information stored in those bytes.

Although you do not need to maintain or use the timestamp portion of the key, you do not need to specify the eight bytes when you create the file. Thus, all key lengths must be defined as at least eight bytes long. If you do not define a user key, the data begins at the eighth byte. If you do supply a user key, it precedes the timestamp. The data begins at the length of your key plus eight bytes. (For more information, see "Creating Queue Files" (page 107)

Figure 15 (page 107) shows the format of each physical record.

Figure 15 Queue File Record Format

User key (optional)	Timestamp (8 bytes)	Data

Creating Queue Files

To create a queue file, use the File Utility Program (FUP) or call the FILE_CREATE_ or FILE_CREATELIST_ procedure. You cannot define partitions or alternate keys for queue files.

When you create a queue file, be sure to leave room for the 8–byte timestamp in the key. If you do not need a user-defined key, specify a key length of eight bytes for the file. Otherwise, specify the length of your key plus eight bytes. In addition, leave room for the 8–byte key in your record length definition.

The key offset value must equal zero. You can omit the key specifier designation; however, if you set it, it must equal zero.

When you create an Enscribe queue file, you can specify the size of the primary and secondary extends. Format 1 files can have from 1 through 65,535 pages (where a page is 2048 bytes). Format 2 files can have from 1 through 536,870,912 pages.

Queue File Examples

These examples illustrate the three different ways to create queue files.

Example 1: Creating a Queue File with FUP

This example shows how you might use FUP to create a queue file:

```
10> FUP
- SET TYPE K
- SET CODE 1001
- SET QUEUEFILE
- SET EXT (20,10)
- SET MAXEXTENTS 64
- SET KEYLEN 10
- SET REC 100
- SET AUDIT
- CREATE QFILE
- EXIT
>
```

Example 2: Creating a Queue File With the FILE_CREATE_ Procedure

This TAL example creates a queue file using the FILE_CREATE_Procedure. The nod name is snot specified in the procedure call, so FILE_CREATE_ obtains the node name from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE. For more information on the =_DEFAULTS DEFINE, see the *TACL Programming Guide*.

```
STRING .QF^Name[0:33] := "$spool.lst.qfile"; ! File name
LITERAL QF^NameLength = 16;                         ! Length in
                                      ! bytes of file name
INT QF^NameLen;
LITERAL Key^Length = 8;                             ! Key length
                                      ! (must be >= 8)
INT Key^Len;
STRING .Key[0:Key^Length - 8];        ! Application key
LITERAL Rec^Len = 100;                ! Record length
INT Error;                            ! Returned error code
?SOURCE $SYSTEM.SYSTEM.EXTDECS0
                        (FILE_CREATE_,FILE_CREATELIST_)
QF^NameLen := QF^NameLength;
Key^Len := Key^Length;

Error := FILE_CREATE_(
    QF^Name:34,            ! filename:maxlen
    QF^NameLen,            ! filenamelen
        1001,              ! filecode
          20,              ! primary-extent-size
          10,              ! secondary-extent-size
          64,              ! maximum-extents
           3,              ! file-type (key-sequenced)
      %000102,             ! options (queue file, audited)
      Rec^Len,             ! recordlen
        4096,              ! blocklen
      Key^Len,             ! keylen
           0);             ! key-offset (must be zero)
IF Error <> 0 THEN ...         ! error
```

Example 3: Creating a Queue File With the FILE_CREATELIST_ Procedure

This example creates a queue file using the FILE_CREATELIST_procedure:

```
! Create a Queue File with FILE_CREATELIST_
STRING .QF^Name[0:33] := "$spool.lst.qfile";              ! File name
LITERAL QF^NameLength = 16;            ! Length in
                                      ! bytes of file name
```

```
            INT QF^NameLen;
            LITERAL Key^Length = 8;                 ! Key length
                                                    ! (must be >= 8)
            INT Key^Len;
            STRING .Key[0:Key^Length - 8];          ! Application key
            LITERAL Rec^Len = 100;                  ! Record length
            INT Items[0:20];                        ! Attribute numbers
            INT Value[0:40];                        ! Attribute values
            INT Error;                              ! Returned error code
            INT Error^Item;                         ! Returned item-in-error
            ?SOURCE $SYSTEM.SYSTEM.EXTDECS0
                                    (FILE_CREATE_,FILE_CREATELIST_)
            QF^NameLen := QF^NameLength;
            Key^Len := Key^Length;
            Items[ 0] := 41;                        ! File type
            Value[ 0] := 3;                         ! = key-sequenced
            Items[ 1] := 42;                        ! File code
            Value[ 1] := 1001;                      ! = 1001
            Items[ 2] := 43;                        ! Logical record length
            Value[ 2] := Rec^Len;                   ! = Rec^Len
            Items[ 3] := 44;                        ! Block length
            Value[ 3] := 4096;                      ! = 4K
            Items[ 4] := 45;                        ! Key offset
            Value[ 4] := 0;                         ! = 0 (unconditional for queue files)
            Items[ 5] := 46;                        ! Key length
            Value[ 5] := Key^Len;                   ! = Key^Len
            Items[ 6] := 48;                        ! Queue File opt. (can also use #71)
            Value[ 6] := 1;                         ! = this is a Queue File
            Items[ 7] := 50;                        ! Primary extent size
            Value[ 7] := 20;                        ! = 20 pages
            Items[ 8] := 51;                        ! Secondary extent size
            Value[ 8] := 10;                        ! = 10 pages
            Items[ 9] := 52;                        ! Maximum extents
            Value[ 9] := 64;                        ! = 64 extents
            Items[10] := 66;                        ! Audited file
            Value[10] := 1;                         ! = audit this file
            Items[11] := 67;                        ! Audit compression
            Value[11] := 1;                         ! = audit compression (can be 0)
            Items[12] := 68;                        ! Data compression
            Value[12] := 0;                         ! = no data compression (can be 1)
            Items[13] := 69;                        ! Index compression
            Value[13] := 0;                         ! = no index compression (can be 1)

            Error := FILE_CREATELIST_(
                QF^Name:34,                         ! filename:maxlen
                QF^NameLen,                         ! filenamelen
                Items,                              ! item-list
                14,                                 ! number-of-items
                Value,                              ! values
                28,                                 ! values-length in bytes
                Error^Item);                        ! returned error-item
            IF Error <> 0 THEN ...                     ! error
```

Accessing Queue Files

Read and write operations to and from queue files differ from that of other key-sequenced files. A write operation is called a queuing operation. When you access a queue file, a read operation typically deletes the record from the file. This operation is called a dequeuing operation. To dequeue a record, use the FILE_READUPDATELOCK64_/READUPDATELOCK[X] procedure. You can also read a record without deleting it; to do this, use the FILE_READ64_/READ[X] procedure.

Specifying Sync-Depth

When you open a queue file, you can specify a sync depth for the file. A non zero sync depth is supported for all operations except for dequeuing; the sync depth for dequeuing operation must equal zero.

If you specify a sync depth that I snot equal to zero for any operation other than dequeuing, the file system attempts to recover from path-related errors (200, 201, 210, and 211). If you specify a sync depth of zero, no path-related recovery is done and the application is responsible for responding to path-related error conditions.

Because of the protection a nonzero sync depth provides, it is generally recommended that a sync depth of one be used for all operations except dequeuing. This implies that multiple opens need to be done for a queue file that is used simultaneously by the same process for enqueuing and dequeuing operations if nonzero sync depth protection is desired. For more information about recovery of retryable errors on queue files, see Communication Path Errors "Communication Path Errors" (page 120).

Queuing a Record

To queue a record, call the FILE_WRITE64_/WRITE[X] procedure to write the record to the file. The disk process sets the timestamp field in the key, which causes the record to be positioned after other existing records that have the same high-order user key.

If the file is audited, the new record is available for read operations when the transaction associated with the write operation commits. If the transaction aborts, the record is never available to read operations.

If the file is not audited, the record is available as soon as the write operation completes successfully.

Unlike other key-sequenced files, a write operation to a queue file will never encounter an error 10 (Duplicate record). This is because all queue file records have unique keys generated for them.

Special "Dummy" Record

When the first record is written to a queue file, the disk process inserts a special dummy record containing all zeros at the front of the file. This record ensures that the file never becomes empty, avoiding the overhead of collapsing the file and then expanding it whenever the last record is dequeued.

The dummy record is never dequeued through FILE_READUPDATELOCK64_/READUPDATELOCK[X]. However, the record is visible under these circumstances:

- A FILE_READ64_, FILE_READLOCK64_, READ[X] or READLOCK[X] operation returns the dummy record.

- The command FUP INFO *filename*, STAT indicates that the record exists.

- The command FUP COPY *filename*, *newlife* causes the dummy record to become a record that can be dequeued by FILE_READUPDATELOCK64_/READUPDATELOCK[X]. Consequently, you should be careful using the FUP COPY command on queue files.

Dequeuing a Record

To dequeue a record, call the FILE_READUPDATELOCK64_/READUPDATELOCK[X] procedure. If the read operation is successful, the disk process deletes the record from the file. If the disk process cannot return a record because the file is empty or there are no unlocked records which meet the selection criteria, the disk process retains the request until the file contains a record that fits the request.

To read a record with the FILE_READUPDATELOCK64_/READUPDATELOCK[X] procedure, these requirements must be met:

- An application must have write access to the file.

- The sync depth must be zero.

- The record must not be locked.

- The record must satisfy any key comparison rules defined by a prior call to the KEYPOSITION procedure. (For more information about positioning, see Dequeuing With Positioning"Dequeuing With Positioning" (page 113).)

The disk process skips over locked records until it reaches an unlocked record that satisfies the record selection rules. The record selection rules are specified by the last invocation of FILE_SETKEY_ or KEYPOSITION. If the application does not call either of these procedures, the disk process considers that all records in the file meet the selection rules. For more information, refer to Dequeuing With Positioning"Dequeuing With Positioning" (page 113).

Records whose user keys are identical for the requested compare length are extracted according to the timestamp generated at the time of insertion. The order of extraction may not be sequential if locked records are encountered. Because each record is deleted after it is read, subsequent calls to FILE_READUPDATELOCK64_/READUPDATELOCK[X] will reposition to the beginning of the range of records which satisfies the selection rules.

The current primary-key value is not updated after the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation.

Example

Figure 16 (page 111) shows how two processes might access a queue file. If process A attempts to dequeue a record while the file is empty, the process waits until the file contains a record.

Figure 16 Dequeuing a Record
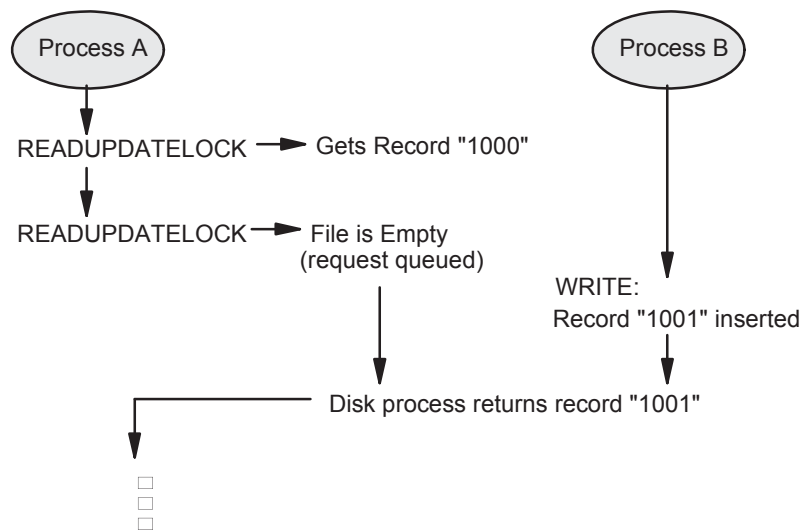
The insertion of a new record causes the waiting Process A to be awakened and presented with the new record.

Dequeuing From Audited Files

If the queue file is audited, the FILE_READUPDATELOCK64_/READUPDATELOCK[X] procedure must be called from within a transaction. An application should commit the transaction only after it processes the data it has extracted from the file, since the

FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation causes the record to be deleted from the file.

If a transaction that is associated with a dequeue request aborts, the disk process reinserts the record at the same logical location from which it came. The abort operation also awakens any processes waiting to dequeue a record from the file.

If the read operation is not successful because the file is empty or all records are locked, the disk process retains the request and waits until one of these events occurs:

- The disk process waits until a transaction completes that includes insertion of a record. When the new record is available, the disk process retries the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation. Note that either of these can make a record available:

  ○ A transaction that commits after inserting a record

  ○ A transaction that aborts after dequeuing a record

  ○ If a special queue-file timeout on the read operation expires, the disk process returns an error 162 (operation timed out) to the requesting process.

One exception is the use of exact positioning. If the application requests exact positioning and the file is empty or the record does not exist, the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation receives an error 11 (record not found) and does not queue the request.

Generally, errors (other than operation timed out) on a FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation should be handled like errors on normal write operations. That is, the transaction should be aborted.

Note the behavior of queue files when generic locking is used. If the lock key-length of a queue file is less than the actual key length, the disk process will perform generic locking on inserted records. Inserting a record when generic locking is enforced will lock existing records that have the same key for the lock key-length. This prevents existing records with the matching generic key from being dequeued until the encompassing transaction completes.

Impact of Records Causing Data Errors

When using audited queue files, there is an additional consideration for error processing. If a dequeuing operation returns a bad record that causes the application to abort the transaction, the bad record is reinserted into the file.

Consider the case where a transaction is started, a record is dequeued, and the contents of the data returned to the application causes it to abort the transaction (either due to a programmatic abort or process failure). The abort operation causes the bad record to be reinserted into the queue file. If the application performs another dequeue operation, it retrieves the same record and could possibly abort again.

Although this might not cause difficulties, the application would not progress past the bad record. To avoid this situation, validate record contents prior to processing data.

This problem only affects audited operations; in the unaudited case, the bad record is not reinserted into the file, but is lost.

Dequeuing Records and Nowait I/O

If the time limit expires prior to the queue file timeout, the FILE_READUPDATELOCK64_/READUPDATELOCK[X] request is canceled if it was a file-specific call (that is, the file number is other than -1). With non file-specific calls, FILE_READUPDATELOCK64_/READUPDATELOCK[X] is not canceled for the queue file. A canceled FILE_READUPDATELOCK64_/READUPDATELOCK[X] can result in the loss of a

record from the queue file. This problem is particularly acute for queue files, since a dequeuing operation can be delayed until the file contains a record that fits the request.

For audited queue files only, your application can recover from a timeout error by calling the ABORTTRANSACTION procedure to ensure that any dequeued records are reinserted into the file. The corresponding transaction must then be restarted.

NOTE: For unaudited queue files, your application should never call AWAITIO[X]/FILE_AWAITIO64_ with a time limit greater than OD if a READUPDATELOCK[X]/FILE_READUPDATELOCK64_ is pending. The recovery procedure described above does not work on unaudited queue files.

Dequeuing From Unaudited Files

If the read operation is not successful because the file is empty or all records are locked, the read is suspended until one of these events occurs:

- The disk process waits until a FILE_WRITE64_/WRITE[X] operation completes successfully.

- If a special queue file timeout on the read operation expires, the disk process returns an error 162 (operation timed out) to the requesting process.

As with audited files, one exception is the use of exact positioning. If the application requests exact positioning and the file is empty or the record does not exist, the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation receives an error 11 (record not found) and does not queue the request.

Examining a Record

To read a record without deleting it, use the FILE_READ64_, FILE_READLOCK64_, READ[X] or READLOCK[X] procedures. These procedures function exactly the same as when used for a key-sequenced file. Note, however, that FILE_READ64_, FILE_READLOCK64_, READ[X] and READLOCK[X] differ from FILE_READUPDATELOCK64_/READUPDATELOCK[X] in these ways:

- The disk process does not delete a record after reading it

- If no record is available, the request is not retained and the disk process returns an error 1 (EOF)

- A read operation returns the entire record, including the 8-byte timestamp. To delete the record after examining it, use exact positioning and then call the FILE_READUPDATELOCK64_/READUPDATELOCK[X] procedure.

Dequeuing With Positioning

To change the access path and positioning dynamically, you can precede calls to FILE_READUPDATELOCK64_/READUPDATELOCK[X] with a call to FILE_SETKEY_ or KEYPOSITION. This method allows an application to move to a random position in the file, establish a subset of records to be retrieved, and apply other selection rules according to the specified parameters.

The disk process does not dequeue locked records. Locked records which exist within the desired key range are skipped until an unlocked record is found or the key in a record is outside of the desired range.

Approximate Positioning

Figure 17 (page 114) illustrates how approximate positioning works for a queue file when the call to FILE_SETKEY_ (or equivalent call to KEYPOSITION) has these parameters:

```
key-value = "MA"
positioningmode = 0
comparelength is omitted = 2
```

The queue file has a key length of 10 bytes, so the user key length is 2 bytes.

Figure 17 Using Approximate Positioning With a Queue File
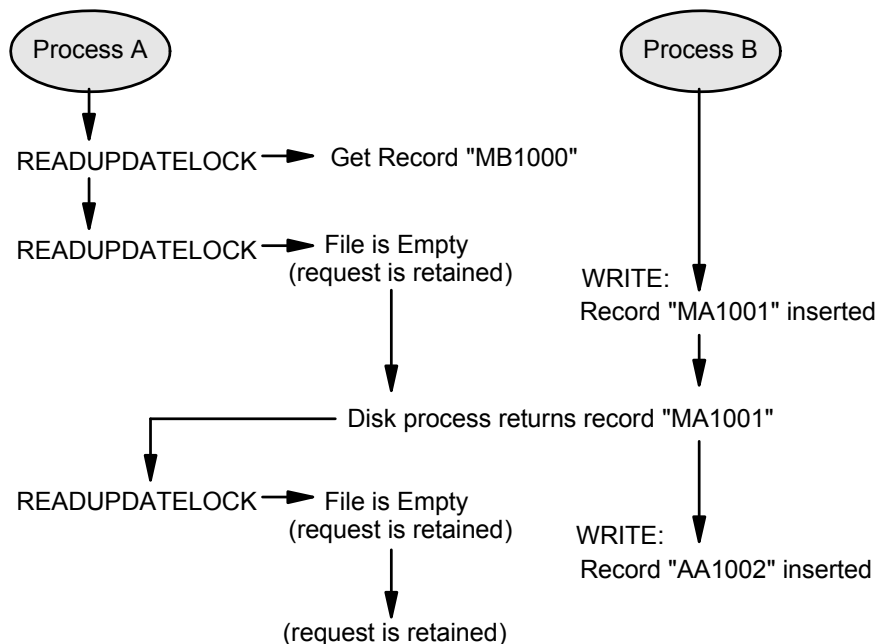
This example shows two special actions of the queue file, compared to a standard key-sequenced file:

- The second READUPDATELOCK call retrieved a record whose key was less than that in the record previously retrieved. This behavior is different than that of an ordinary key-sequenced file.

- The second insertion in the file (key = "AA") does not cause process A to be awakened, because the key of the inserted record does not match the selection criteria established by the KEYPOSITION call.

Generic Positioning

illustrates how approximate positioning works for a queue file when the call to FILE_SETKEY_ (or equivalent call to KEYPOSITION) has these parameters:

```
key-value = "TA"
comparelength = 2
key-value-len = 2
positioningmode = 1
```

The queue file has a key length of 10 bytes, so the user key length is 2 bytes.

Figure 18 Using Generic Positioning With a Queue File
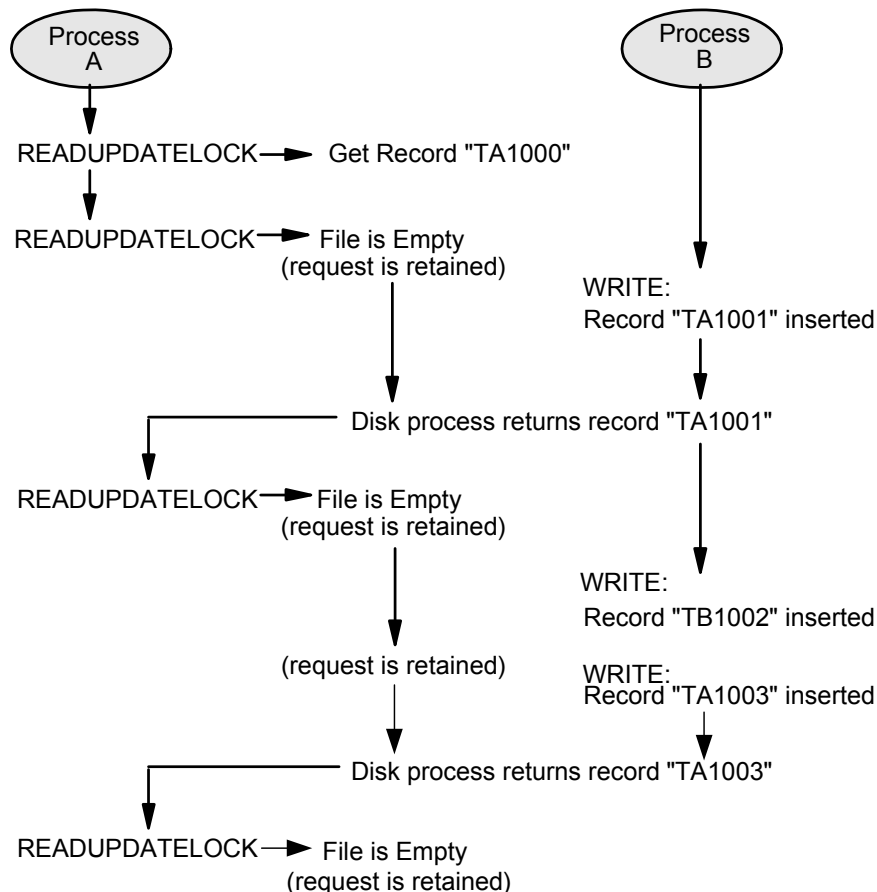
Note that the second insertion does not cause Process A to be awakened, because the user key in the new record does not satisfy the key selection criteria. The third insertion does satisfy the criteria, however, so the disk process returns the record to Process A and reawakens Process A.

This example also illustrates how an application could use multiple servers or dequeuing processes to read from the same file. Each server could access a subset of the file as designated by the high-order field of the primary key (in this case, records with "TA" in the high-order key are being dequeued by process A). In this manner, the high-order key file can be used to logically partition the file across multiple servers.

The high-order key field can also be used to specify relative priority of a queued record. Records with smaller valued keys are positioned before those of higher values. Thus, a server that always reads from the start of the file will dequeue records in ascending key order. This permits prioritization of records within the Queue File.

Exact Positioning

The use of exact positioning is not typical for queue files, since applications usually access queue files in a specified order rather than by exact key value. You can, however, use exact positioning for a queue file. For example, you might use the FILE_READ64_/READ[X] procedure to access specific records within the file without changing their placement in the queue file, and then delete specific records after examining them.

Using the Current Key

The current key for a queue file has meaning only after a FILE_READ64_/READ[X] operation. Unlike standard key-sequenced files, you cannot assume the current key is accurate after the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation.

This behavior affects the operation of the FILE_GETINFOLIST_ and FILERECINFO operations. The FILE_GETINFOLIST_ procedure does not return a meaningful current key value after a FILE_WRITE64_, WRITE[X], FILE_READUPDATELOCK64_ or READUPDATELOCK[X] operation. It can, however, return a current key value after a FILE_READ64_/READ[X] operation. The FILERECINFO procedure does not return the current key value for queue files.

Specifying Timeout Periods

To specify a timeout period for read operations on queue files, use SETMODE function 128. Otherwise, a default timeout period of 60 seconds applies. The purpose of the timeout period is to limit the time spent on dequeue operations, especially for audited files. If the read operation is not completed within the timeout period, an error 162 (operation timed out) is returned.

The parameters for the SETMODE functions 128 are:

param1   The high-order word of the timeout value (in units of one hundredth of a second).

param2   The low-order word of the timeout value (in units of one hundredth of a second).

The two words are combined to form a 32-bit integer for the timeout value. These values are reserved:

-2D   Default timeout period (60 seconds).

-1D   Infinite timeout period (timeout error is not returned.)

0D   No timeout period (error is returned immediately if record cannot be read.)

NOTE: Do not use the timeout option of AWAITIO[X]/FILE_AWAITIO64_ to complete a READUPDATELOCK[X]/FILE_READUPDATELOCK64_ operation. The cancellation that occurs after the timeout expires hides the fact that a record may have been dequeued from the file.

Locking a Record

To lock a record, perform a READLOCK[X]/FILE_READLOCK64_ operation. For audited files, any records associated with an uncommitted transaction are also considered locked.

Network Considerations

You cannot access a queue file from a system running an operating system prior to D20.

Performance Considerations

Although multiple servers (dequeuing processes) can be used on a Queue File, there are practical limits on how many processes should be used. Each time a transaction completes (for audited Queue Files) or record is inserted (for non-audited Queue Files), the Disk Process will re-execute each FILE_READUPDATELOCK64_/READUPDATELOCK operation which is waiting for a new record. When there are many FILE_READUPDATELOCK64_/READUPDATELOCK operations waiting, the overhead to perform this action can become excessive, and reduce the throughput on the Queue File and/or affect applications sharing the CPU with the primary Disk Process. The amount of impact is a factor on the number of read processes waiting at a given time, and the speed of the CPU.

As a general guideline, do not use more than a few dozen dequeueing processes. Although supported, having more than 100 processes for a Queue File can had adverse affects on a system. As an alternative, consider using multiple queue files.

Access Examples

These examples show how to access queue files. For brevity, the timestamps shown in the examples have been truncated to 4-byte numeric strings (for example, "1001"). If the key used in the example is longer than eight bytes, the key is displayed as "<user-key><timestamp>"for example, "AB1001").

Example 1: Opening a Queue File

This TAL example opens a queue file:

```
INT QF^Num;           ! Queue File number
STRING .QF^Name[0:33] := "$spool.lkp.qfile"; ! File name
LITERAL QF^NameLength = 16; ! Length in bytes of file name
INT Error;                 ! Returned error code

?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (FILE_OPEN_)
!
! Open the Queue File
!Key^Len := Key^Length;

Error := FILE_OPEN_(
  QF^Name:QF^NameLength,  ! filename:length
  QF^Num,                 ! filenum
  0,                      ! access = read/write (can be > 0)
  0,                      ! exclusion = shared (can be > 0)
  0,                      ! nowait-depth = 0 (can be > 0)
  0);                     ! sync-depth (must be 0 if
                          ! READUPDATELOCK is used)
```

Example 2: Enqueuing a Record

This example shows a TAL procedure that inserts a record into a queue file:

```
INT QF^Num;                  ! Queue File number
STRING .QF^Name[0:33] := "$spool.lkp.qfile"; ! File name
LITERAL QF^NameLength = 16;    ! Length in bytes of file name
INT QF^NameLen;
LITERAL Key^Length = 8;        ! Key length (must be >= 8)
INT Key^Len;
STRING .Key[0:Key^Length - 8];! Application key
LITERAL Rec^Len = 100;         ! Record length
INT Byte^Count;                ! Number of bytes read/written
INT Error;                     ! Returned error code
FIXED Trans^Tag;               ! Transaction ID
STRING .Buffer[0:Rec^Len - 1]; ! Record buffer
STRING .Data[0:Rec^Len - Key^Length - 1]; ! Data

?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (WRITE, BEGINTRANSACTION) ?SOURCE
$SYSTEM.SYSTEM.EXTDECS0 (ENDTRANSACTION)

! Enqueue a record into a Queue File
INT PROC ENQUEUE(Error);
  INT .Error;                          ! Error code
  BEGIN
  Buffer[0] ':=' Key FOR Key^Len - 8;  ! Move key into front
                                       ! of buffer
  Buffer[Key^Len] ':=' Data FOR Rec^Len - Key^Len;
                                       ! Move data record
                                       ! into buffer
  Error := BEGINTRANSACTION(Trans^Tag);! Start a transaction
  CALL WRITE(QF^Num, Buffer, Rec^Len, Byte^Count);
                                       ! Write the record
  IF = THEN
    Error := 0                         ! Clear error code
  ELSE
```

```
        CALL FILEINFO(QF^Num, Error);        ! Obtain error code
    CALL ENDTRANSACTION;                      ! End the transaction
  RETURN Error <> 0;
  END;
```

Example 3: Dequeuing a Record

This TAL example dequeues the first record from a queue file. Note that the queue file must be opened before calling this procedure.

```
INT QF^Num;                ! Queue File number
STRING .QF^Name[0:33] := "$spool.lkp.qfile"; ! File name
LITERAL QF^NameLength = 16;      ! Length in bytes of file name
INT QF^NameLen;
LITERAL Key^Length = 8;          ! Key length (must be >= 8)
INT Key^Len;
STRING .Key[0:Key^Length - 8];   ! Application key
LITERAL Rec^Len = 100;           ! Record length
INT Byte^Count;                  ! Number of bytes read/written
INT Error;                       ! Returned error code
FIXED Trans^Tag;                 ! Transaction identifier
STRING .Buffer[0:Rec^Len-1];     ! Record buffer
STRING .Data[0:Rec^Len - Key^Length - 1];
                                 ! Data being dequeued

?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (KEYPOSITION,
?                                ABORTTRANSACTION,
?                                BEGINTRANSACTION,
?                                ENDTRANSACTION,
?                                FILEINFO,
?                                READUPDATELOCK)
!
! Dequeue the first record from a Queue File
!
INT PROC DEQUEUE(Error);    ! Returns # bytes in queue
                            ! entry, or
  INT .Error;               ! returned error code (or zero)
  BEGIN                     ! -1 if an error occurred.
  Error := 0;               ! Clear error code
  ! Position to beginning of file
  CALL KEYPOSITION(
       QF^Num,         ! filenum
       Key,            ! key-value (not used)
       ,               ! key-specifier (not needed)
       0,              ! length-word = zero (position to start)
       0);             ! positioning-mode = approximate
  DO
    BEGIN
    Error := BEGINTRANSACTION(Trans^Tag); ! Start trans.
    CALL READUPDATELOCK(QF^Num, Buffer, Rec^Len, Byte^Count);
                                   ! Read the first record
    IF = THEN                      ! Check for errors
     BEGIN                         ! No error occurred
     Data[0] ':=' Buffer[Key^Len] FOR Byte^Count - Key^Len;
     RETURN Byte^Count;            ! Extract data and return
     ! Note: An ENDTRANSACTION call should be executed when
     ! the current queue record has been processed.
     END;
  !
  ! Process READUPDATELOCK error
  !
  ! Determine which error occurred
  status := FILE_GETINFO_ (QF^Num, Error);
  IF Error = 162 then      ! Timeout occurred
    BEGIN
```

```
            status := ENDTRANSACTION;   ! Release this transaction
         END
      ELSE                         ! Some other error
         BEGIN
            status := ABORTTRANSACTION; !abort this transaction
            RETURN -1;
         END;
      END                          ! Execute one dequeue operation
UNTIL 0;
END;
```

Example 4: Using KEYPOSITION for Generic Positioning

In this example, the queue file has a key-length of 10 bytes, so the user-key length is 2 bytes. Note that you must open the queue file and specify a key range in key before calling GET^GENERIC.

```
INT QF^Num;                   ! Queue File number
STRING .QF^Name[0:33] := "$pub1.spl1111.qfile"; ! File name
LITERAL QF^NameLength = 19;        ! Length in bytes of file name
INT QF^NameLen;
LITERAL Key^Length = 10;           ! Key length (must be >= 8)
INT Key^Len;
STRING .Key[0:Key^Length - 9];    ! Application key
LITERAL Rec^Len = 100;             ! Record length
INT Byte^Count;                    ! Number of bytes read/written
INT Length^Word;                   ! length-word parameter to
                                   ! KEYPOSITION
INT Error;                         ! Returned error code
INT Error^Item;                    ! Returned item-in-error
FIXED Trans^Tag;                   ! Transaction identifier
STRING .Buffer[0:Rec^Len - 1];    ! Record buffer
STRING .Data[0:Rec^Len - Key^Length - 1];
                                   ! Data being enqueued/dequeued
!
?SOURCE $SYSTEM.SYSTEM.EXTDECS0 (FILE_OPEN_,
?                                READ,
?                                WRITE,
?                                KEYPOSITION
?                                FILE_CREATE_,
?                                FILE_CREATELIST_
?                                ABORTTRANSACTION,
?                                BEGINTRANSACTION,
?                                ENDTRANSACTION
?                                FILEINFO,
?                                READUPDATELOCK,
?                                MYTERM,
?                                STOP,
?                                DEBUG
?                                OPEN,
?                                CLOSE);
!
! Dequeue a record within a generic key range
!
INT PROC GET^GENERIC(Error);        ! Returns # bytes in queue
                                    ! entry, returned
 INT .Error;                        ! error code, zero, or
 BEGIN                              ! -1 if an error occurred.
 Error := 0;                        ! Clear error code
 Length^Word := 2;                  ! Select key-length = 2, and
 Length^Word.<0:7> := 2;            ! compare-length = 2.

 CALL KEYPOSITION(
     QF^Num,                 ! filenum
     Key,                    ! key-value
```

```
            0,                       ! key-specifier (must be zero)
         Length^Word,                ! length-word
         1);                         ! positioning-mode = generic
      ! Note: In practice, it is only necessary to call
      ! KEYPOSITION once to establish the desired key range.
       DO
         BEGIN
         Error := BEGINTRANSACTION(Trans^Tag); ! Start trans.
         CALL READUPDATELOCK(QF^Num, Buffer, Rec^Len, Byte^Count);
                                    ! Read the next record
    IF = THEN                       ! Check for errors
         BEGIN                      ! No error occurred
          Data[0] ':=' Buffer[Key^Len] FOR Byte^Count - Key^Len;
          RETURN Byte^Count;        ! Extract data and return
           ! Note: An ENDTRANSACTION call should be executed when
           ! the current queue record has been processed.
          END;
      !
      ! Process the READUPDATELOCK error
      !

      ! Determine which error occurred
      status := FILE_GETINFO_ (QF^Num, Error);
      IF Error = 162 then       ! Timeout occurred
       BEGIN
          status := ENDTRANSACTION;   ! Release this transaction
       END
      ELSE                        ! Some other error
        status := ABORTTRANSACTION;   ! Abort this transaction
        RETURN -1;
       END                        ! Execute one dequeue operation
      UNTIL 0;
      END;                        ! PROC Get^Generic
```

Communication Path Errors

For queue file errors in the range of 200 through 211 (path loss, takeover, and CPU failure), and for network path errors in the range of 246 through 249, the file system attempts recovery only if the sync depth of the open file is greater than zero. However, dequeuing operations (FILE_READUPDATELOCK64_/READUPDATELOCK[X] calls) require a sync depth of zero; therefore, the file system cannot recover dequeuing failures due to path-related errors.

A process that only writes to (that is, it does not dequeue from) a queue file can ensure a recovery attempt by setting the sync depth to one or greater (depending on how many outstanding FILE_WRITE64_/WRITE[X] operations it intends to perform).

If the sync depth is zero but the file is audited, an application can recover from a path loss condition by aborting the pending transaction.

Processes that dequeue (call FILE_READUPDATELOCK64_/READUPDATELOCK[X]) from an audited queue file can invoke ABORTTRANSACTION and retry the FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation. This will ensure that if a record was deleted prior to the failure, it will be reinserted. Note, however, that all records modified during the current transaction will be restored. Therefore, the application must be capable of retrying the entire aborted transaction from the beginning.

Processes that access unaudited queue files have little means of recovery from path-related errors. There are two possible error conditions that could result in lost or extra records in a queue file:

- A FILE_READUPDATELOCK64_/READUPDATELOCK[X] operation could encounter a path-related error after a record was deleted. In this case, the contents of the record would be lost.

- A FILE_WRITE64_/WRITE[X] operation that encounters a path-related error might have successfully inserted a record into the file. An attempt to retry the same FILE_WRITE64_/WRITE[X] operation would result in the insertion of a second copy of the same record with a different unique key value (different timestamp value).

In summary, if you require protection from path loss conditions, use audited files.

8 Entry-Sequenced Files

Enscribe Entry-Sequenced Files

Enscribe entry-sequenced files are designed for sequential access. They consist of variable-length records that are always appended to the end of the file; as a result, the records in the file are arranged physically in the order in which they were added to the file.

Figure 19 (page 123) illustrates the structure of an entry-sequenced file.

The primary key of an entry-sequenced file consists of a record's block number (within the file) and its record number (within the block). When used with FILE_SETPOSITION_, the key is an 8-byte value in which the block number occupies the leftmost 4 bytes of the key and the record number occupies the rightmost 4 bytes. When used with POSITION, the key is a 4-byte value whose format depends on the file's block size, as follows:

Block size	Number of bits for block	Number of bits for record
4096	20	12
2048	21	11
1024	22	10
512	23	9

The record's address is typically used and manipulated internally by the file system, and there is usually no reason for you to know its value. You can, however, obtain the address of the record just read or written by using the FILE_GETINFOLIST_ system procedure.

Applicable System Procedures

You use these system procedures to create and access Enscribe entry-sequenced files:

- FILE_CREATE_, FILE_CREATELIST_
- FILE_OPEN_, FILE_CLOSE_, AWAITIO[X], FILE_AWAITIO64_
- FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, LOCKFILE, UNLOCKFILE, LOCKREC, UNLOCKREC
- FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X], READUPDATELOCK[X]
- FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITE[X], WRITEUPDATE[X], WRITEUPDATEUNLOCK[X]
- FILE_SETKEY_, FILE_SAVEPOSITION_, KEYPOSITION, POSITION, FILE_SETPOSITION_
- FILE_GETINFO_, FILE_GETINFOLIST_, FILE_GETINFOLISTBYNAME_, FILE_GETINFOBYNAME_
- SETMODE, CONTROL, FILE_CONTROL64_

Types of Access

After creating the file, there are essentially three operations that you can perform:

1. Use FILE_WRITE64_/WRITE calls to add new records to the end of the file.
2. Use FILE_READ64_/READ calls to retrieve records from the file.
3. Use FILE_SETKEY_ or KEYPOSITION calls to specify an alternate-key access path and then use FILE_READ64_/READ calls to retrieve records that contain the specified alternate-key value.

You can also use the file and record locking system procedures FILE_LOCKFILE64_,
FILE_LOCKREC64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, FILE_READLOCK64_,
LOCKFILE, UNLOCKFILE, READLOCK, LOCKREC, and UNLOCKREC.Enscribe entry-sequenced
files are not designed for random access. If the data records contain unique alternate-key values,
however, you can use KEYPOSITION in conjunction with FILE_READ64_, FILE_READLOCK64_,
READ or READLOCK to access them randomly.

NOTE: If an error occurs during an attempt to insert a record into an alternate-key file, 0-length
records might occur in the primary entry-sequenced file. This can also occur when a transaction
aborts that inserted records into an audited entry-sequenced file. The 0-length records are
substituted for the inserted records during TMF BACKOUT.

Figure 19 Entry-Sequenced File Structure

Creating Entry-Sequenced Files

You create Enscribe entry-sequenced files by using either the File Utility Program (FUP) or by
calling either the FILE_CREATE_ or the FILE_CREATLIST_ system procedures. If you wish to
explicitly create key-sequenced format 2 file, use the procedure call FILE_CREATELIST_ with
item code 195.

When you create a partitioned (multivolume) file, the file system automatically creates all of the
partitions of that file when the first partition is created.

If you use a system procedure to create an entry-sequenced file and the file contains alternate-key
fields, then you must also create one or more alternate-key files. If you are using FUP to create
the primary-key file, however, FUP automatically creates any required alternate-key files.

When creating an entry-sequenced file, you must consider the maximum logical record size, the
data block length, and disk extent sizes.

NOTE: Since the primary keys of entry-sequenced files are larger in format 2 files, the size of
alternate key records will also be larger. This may affect code that creates alternate-key files and
programs that directly read the contents of alternate-key files may be affected

Logical Records

A record is the unit of information transferred between an application program and the file system.

When creating an entry-sequenced file, you must specify the maximum logical record size of that
file. The particular maximum record size that you choose when creating a file depends upon the
particular requirements of your application.

For entry-sequenced files, the maximum length of a logical record is 24 bytes (or 48 bytes for
format 2 files) less than the block size. The maximum allowed block size for format 2 files is 4
KB.

The data records that you write to an entry-sequenced file can be of varying lengths, but none
can exceed the maximum logical record size specified when the file was created. If you try to
write a record that is longer than the defined maximum record length, the file system rejects the
operation and returns an error 21 (illegal count).

Blocks

A block is the unit of information transferred between the disk process and the disk. A block
consists of one or more logical records and, in the case of entry-sequenced files, associated
control information. This control information, which is used only by the system, is summarized
in "Block Formats of Structured Files" (page 172).

The block size of an Enscribe entry-sequenced file must be 512 bytes, 1 KB, 2 KB, or 4 KB.

The block size must include 22 bytes (format 1 files) and 44 bytes (format 2 files) per block for block control information and 2 or 4 bytes per record for record control information. Therefore, the maximum number of records that you can store in each block is

Format 1 Files

```
N = ( block-size - 22 ) / ( record-size + 2 )
```

Format 2 Files

```
N = ( block-size - 44) / ( record-size + 4 )
```

If records are of varying lengths, then N is the average number of records per block and record-size is the average record length.

Regardless of the record length, the maximum number of records that can be stored in a single block is 511 for a format 1 file.

A record cannot span block boundaries (that is, it cannot begin in one block and end in another). Therefore, the block size for an entry-sequenced file must be at least `record-length` + 2 + 22 bytes for format 1 files and `record-length` + 4 + 44 bytes for format 2 files.

Disk Extent Size

When you create an Enscribe entry-sequenced file, you can specify:

- The size of the primary and secondary extents. Format 1 files can have from 1 through 65,535 pages (where a page is 2048 bytes), while format 2 files can have from 1 through 536,870,912 pages.

- The maximum number of extents to be allocated for the file (16 or more for nonpartitioned entry-sequenced files).

If you do not specify extent sizes, both the primary and secondary extent sizes default to one page.

If you do not specify the maximum number of extents, MAXEXTENTS defaults to 16.

For nonpartitioned entry-sequenced files, you can change the MAXEXTENTS value dynamically during program execution by using either the SETMODE 92 system procedure or the FUP ALTER command.

File Creation Examples

The pages that follow present annotated examples showing how to create:

1. An entry-sequenced file
2. An entry-sequenced file with alternate keys
3. An alternate-key file
4. A partitioned entry-sequenced file

Example 1: Creating an Entry-Sequenced File

This example shows how to create a file for logging summary records of financial transactions as they occur. Because the records will always be written to the file sequentially in the order in which they are generated, it is reasonable to use an entry-sequenced file for storing them.

Assume that the desired record format is:

Byte Offset:

0	10	16	24

transid	date	time-stamp	account-number

40	45	55	60

trans-type	amount-transacted	terminal-number
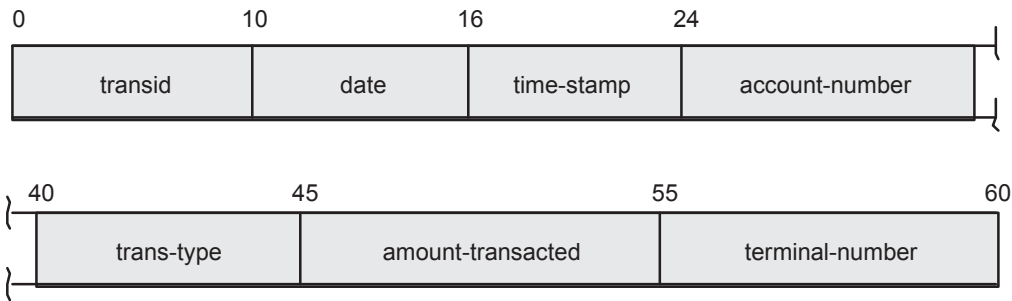
With a record size of 60, selecting a block size of 4096 results in a blocking factor of 65 records per block:

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

65 = (4096 - 22) / (60 + 2)
```

If you designate the primary extent size as 1000 pages and the secondary extent size as 500 pages, then the primary extent can accommodate 32,500 transaction summary records and each secondary extent can accommodate 16,250 transaction summary records. If all 16 extents are eventually used, the file will accommodate a total of 276,250 transaction summary records

You could create the file by using these FUP commands:

```
>volume $store1.svol1
>fup
-set type e
-set ext (1000,500)
-set rec 60
-set block 4096
-show
    TYPE E
    EXT ( 1000 PAGES, 500 PAGES )
    REC 60
    BLOCK 4096
-create tranfile
CREATED - $STORE1.SVOL1.TRANFILE
```

Using the FILE_CREATE_ system procedure, you could create the file by including this TAL code in one of your application modules:

```
LITERAL name^length = 22,
        pri^extent = 1000,
        sec^extent = 500,
        rec^len = 60,
        data^block^len = 4096,
        file^type = 2; ! entry-sequenced
INT filenum;
INT error;
INT namelen;
STRING .filename [0:21] := "$STORE1.SVOL1.TRANFILE";
?NOLIST
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                                FILE_OPEN_,
?                                FILE_CREATE_,
?                                READ)
?LIST
PROC DO^THE^WORK MAIN;
BEGIN
   namelen := name^length;
   ERROR := FILE_CREATE_ (filename:name^length,namelen,,
                  pri^extent, sec^extent,, file^type,, rec^len,
                  data^block^len);
```

```
        ERROR := FILE_OPEN_ (filename:name^length, filenum);
        ERROR := FILE_CLOSE_ (filenum);
   END;
```

Example 2: Creating an Entry-Sequenced File With Alternate Keys

This example shows how to create the file illustrated in , but defines the terminal-number field as an alternate key.
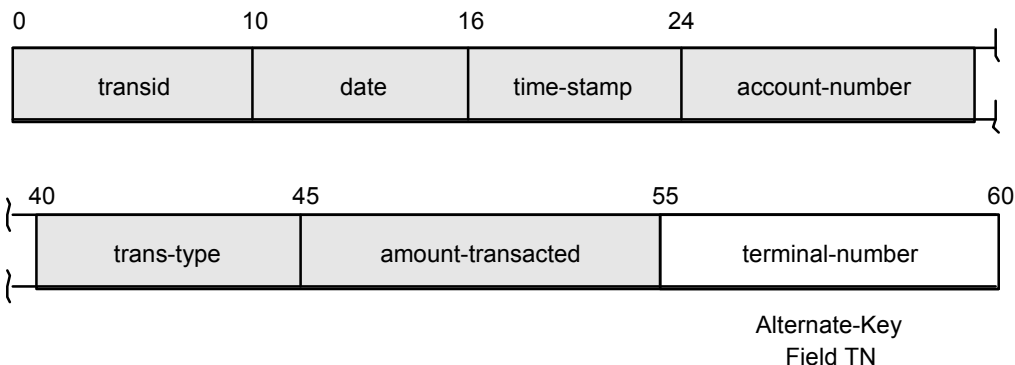
Byte Offset:

You could create the file by using these FUP commands:

```
>volume $store1.svol1
>fup
-set type e
-set ext (1000,500)
-set rec 60
-set block 4096
-set altkey ("tn",keyoff 55,keylen 5)
-set altfile (0,alttran)
-show
     TYPE E
     EXT ( 1000 PAGES, 500 PAGES )
     REC 60
     BLOCK 4096
     ALTKEY ( "TN", FILE 0, KEYOFF 55, KEYLEN 5 )
     ALTFILE ( 0, $STORE1.SVOL1.ALTTRAN )
     ALTCREATE
-create tranfile
CREATED - $STORE1.SVOL1.TRANFILE
CREATED - $STORE1.SVOL1.ALTTRAN
```

Using the FILE_CREATE_ system procedure, you could create the file by including this TAL code in one of your application modules:

```
LITERAL name^length = 22,
        num^altkeys = 1,
        num^altkey^files = 1,
        item^list^len = 10;
INT error;
INT error2;
INT namelen;
STRING .filename [0:name^length-1] :=
                "$STORE1.SVOL1.TRANFILE";

INT .item^list [0:item^list^len-1];

STRUCT value^list;
  BEGIN
    INT file^type;
    INT logical^reclen;
```

```
        INT block^length;
        INT pri^extent;
        INT sec^extent;
        INT altkeys;
        STRUCT altkey^descr [0:num^altkeys-1];
          BEGIN
            STRING key^specifier [0:1];
            INT key^length;
            INT key^offset;
            INT key^filenum;
            INT null^value;
            INT attributes;
          END;
        INT num^alt^key^files;
        STRUCT name^length^info [0:num^altkey^files-1];
          BEGIN
            INT file^name^len;
          END;
        STRING file^names [0:20];
      END;
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                                   FILE_OPEN_,
?                                   FILE_CREATELIST_,
?                                   READ)
?LIST
PROC DO^THE^WORK MAIN;
BEGIN
  namelen := name^length;

  item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                  ZSYS^VAL^FCREAT^LOGICALRECLEN,
                  ZSYS^VAL^FCREAT^BLOCKLEN,
                  ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                  ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                  ZSYS^VAL^FCREAT^NUMALTKEYS,
                  ZSYS^VAL^FCREAT^ALTKEYDESC,
                  ZSYS^VAL^FCREAT^NUMALTKEYFILES,
                  ZSYS^VAL^FCREAT^ALTFILELEN,
                  ZSYS^VAL^FCREAT^ALTFILENAMES ];
  value^list.file^type := 2; ! entry-sequenced
  value^list.logical^reclen := 60;
  value^list.block^length := 4096;
  value^list.pri^extent := 1000;
  value^list.sec^extent := 500;
  value^list.altkeys := num^altkeys;
  value^list.altkey^descr[0].key^specifier ':=' "TN";
  value^list.altkey^descr[0].key^length := 5;
  value^list.altkey^descr[0].key^offset := 55;
  value^list.altkey^descr[0].key^filenum := 0;
  value^list.altkey^descr[0].null^value := 0;
  value^list.altkey^descr[0].attributes := 0;
  value^list.num^alt^key^files := num^altkey^files;
  value^list.name^length^info[0].file^name^len := 21;
  value^list.file^names ':=' "$STORE1.SVOL1.ALTTRAN";

  ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
           item^list, item^list^len, value^list,
           $LEN(value^list), error2);
END;
```

When you use a system procedure to create your entry-sequenced file, you must create your alternate key files separately, as shown in the next example.

Example 3: Creating an Alternate-Key File Programmatically

When you use FUP to create the primary file, FUP automatically creates any required alternate-key files. If you create the primary file programmatically, however, you must create the alternate-key file yourself as a separate operation.

You could create the alternate-key file for the previous example (Example 2) by including this TAL code in one of your application modules:

```
LITERAL name^length = 21,
        pri^extent = 30,
        sec^extent = 15,
        file^type = 3,
        rec^len = 11,
        data^block^len = 4096,
        key^length = 11, ! maximum alt.-key length
                         ! + big files primary-key length
                         ! + 2
        key^offset = 0;

INT error;
INT namelen;
STRING .filename [0:name^length-1]
        := "$STORE1.SVOL1.ALTTRAN";

namelen := name^length;

error := FILE_CREATE_(filename:name^length, namelen,,
                      pri^extent, sec^extent,, file^type,, rec^len,
                      data^block^len, key^length, key^offset);

IF Error <> 0 THEN ... ! error
```

Example 4: Creating a Partitioned Entry-Sequenced File

This example shows how to create the file illustrated in Example 1: Creating an Entry- Sequenced File, but enables it to ultimately span four partitions.

You could create the file by using these FUP commands:

```
>volume $store1.svol1
>fup
-set type e
-set ext (1000,500)
-set rec 60
-set block 4096
-set part (1,$store2,1000,500)
-set part (2,$store3,1000,500)
-set part (3,$store4,1000,500)
-show
    TYPE E
    EXT ( 1000 PAGES, 500 PAGES )
    REC 60
    BLOCK 4096
    PART ( 1, $STORE2, 1000, 500 )
    PART ( 2, $STORE3, 1000, 500 )
    PART ( 3, $STORE4, 1000, 500 )
-create tranfile
CREATED - $STORE1.SVOL1.TRANFILE
```

Note that each partition must reside on a separate disk volume. Within those volumes, however, the partitions all have the same subvolume name and file name (SVOL1.TRANFILE in this example). All four partitions are created at the same time.

When all 16 extents of the primary partition (#0) have been entirely used, the file system automatically begins using partition #1; when all 16 extents of that partition have been entirely used, the file system then begins using partition #2; and so forth.

Using the FILE_CREATELIST_ system procedure, you could create the file by including this TAL code in one of your application modules:

```
LITERAL name^length = 22,
        num^partitions = 3,
        item^list^len = 9;

INT error;
INT error2;
INT namelen;
STRING .filename [0:name^length-1] :=
        "$STORE1.SVOL1.TRANFILE";

INT .item^list [0:item^list^len-1];

STRUCT value^list;
   BEGIN
     INT file^type;
     INT logical^reclen;
     INT block^length;    INT pri^extent;
     INT sec^extent;
     INT partitions;
     STRUCT part^info [0:num^partitions-1];
        BEGIN
           INT part^pri^extent;
           INT part^sec^extent;
        END;
     STRUCT vol^name^len [0:num^partitions-1];
        BEGIN
           INT vol^name^act^len;
        END;
     STRING vol^names [0:21];
   END;
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                                 FILE_OPEN_,
?                                 FILE_CREATELIST_,
?                                 READ)
?LIST
PROC DO^THE^WORK MAIN;
BEGIN
    namelen := name^length;

    item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                    ZSYS^VAL^FCREAT^LOGICALRECLEN,
                    ZSYS^VAL^FCREAT^BLOCKLEN,
                    ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                    ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                    ZSYS^VAL^FCREAT^NUMPRTNS,
                    ZSYS^VAL^FCREAT^PRTNDESC,
                    ZSYS^VAL^FCREAT^PRTNVOLLEN,
                    ZSYS^VAL^FCREAT^PRTNVOLNAMES];

value^list.file^type := 2;              ! entry-sequenced
value^list.logical^reclen := 60;
value^list.block^length := 4096;
value^list.pri^extent := 1000;
value^list.sec^extent := 500;
value^list.partitions := 3;
value^list.part^info[0].part^pri^extent := 1000;
value^list.part^info[0].part^sec^extent := 500;
value^list.part^info[1].part^pri^extent := 1000;
value^list.part^info[1].part^sec^extent := 500;
```

```
value^list.part^info[2].part^pri^extent := 1000;
value^list.part^info[2].part^sec^extent := 500;
value^list.vol^name^len.vol^name^act^len[0] := 7;
value^list.vol^name^len.vol^name^act^len[1] := 7;
value^list.vol^name^len.vol^name^act^len[2] := 7;
value^list.vol^names ':=' "$STORE2$STORE3$STORE4";

ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
        item^list, item^list^len, value^list,
      $LEN(value^list), error2);
END;
```

Accessing Entry-Sequenced Files

You can perform three basic operations with an entry-sequenced file:

- Add (FILE_WRITE64_/WRITE) records to the end of the file.

- Read (FILE_READ64_, FILE_READLOCK64_, READ or READLOCK) records sequentially according to either their primary keys or a specified alternate-key value.

- Specify the desired alternate-key access path (FILE_SETKEY_ or KEYPOSITION).

In addition, you can use FILE_GETINFOLIST_ to obtain the address of the record just read or written, and you can use FILE_SETKEY_ or KEYPOSITION in conjunction with unique alternate-key values to read individual records in a random manner.

Sequential Access

Enscribe entry-sequenced files are designed for sequential access. You always append new data records to the end of the file using the FILE_WRITE64_/WRITE system procedure

When reading an entry-sequenced file, you generally do so in a sequential manner either by primary key (all records in the file) or by a particular alternate-key value (all records in the file that contain that value in the designated field).

When you open the file, the access path is by primary key. You use the FILE_SETKEY_ or KEYPOSITION system procedure to change the access path from the primary key to a particular alternate key and from one alternate key to another. After reading data records by a particular alternate key, you can reset the access path back to the primary key (starting at the beginning of the file) by using FILE_SETKEY_ or KEYPOSITION with both a key specifier and a key value of 0.

If the data records are of variable lengths, you specify the maximum record length as the read-count parameter in the FILE_READ64_, FILE_READLOCK64_, READ or READLOCK procedure call. The file system recognizes the end of the physical record on the disk and returns the actual data record length, in bytes, as the *count-read* parameter.

Random Access

If every data record in an entry-sequenced file contains a unique alternate-key value in a particular field, you can use FILE_SETKEY_ or KEYPOSITION in conjunction with FILE_READ64_, FILE_READLOCK64_, READ or READLOCK to read individual data records in a random manner.

For example, if the file contains transaction summary records and one of the fields in each data record contains a unique transaction number, you could use that field as an alternate key to locate any desired transaction record.

Access Examples

The remainder of this section presents annotated examples illustrating the most common ways to access Enscribe entry-sequenced files.

Example 1. Writing to an Entry-Sequenced File

To append a new data record to the end of an entry-sequenced file you use WRITE or WRITEX to add your data record to the file.

```
CALL WRITE (filenum, buffer, write^count);
IF <> THEN ... ! error
```

If you need to obtain the actual record address of the newly appended data record, use FILE_GETINFOLIST_:

```
error := FILE_GETINFOLIST_ (filenum, itemlist, 1, result);
```

where *itemlist* is defined as:

```
itemlist := 12     ! Return address of current record
```

Example 2. Reading Sequentially by Primary Key

Assume that the particular entry-sequenced file you are working with contains variable length transaction summary records, the largest of which is 200 bytes in length. To read the entire file sequentially, you merely open the file and then repeatedly call the READ system procedure until you encounter the EOF mark.

```
error := FILE_OPEN_ (filename:length, filenum, ... );

read^count := 200;
eof := 0;
WHILE NOT eof DO
   BEGIN ! read loop
     CALL READ (filenum, buffer, read^count, count^read);
     IF > THEN eof := 1
      ELSE
        IF < THEN ... ! error
        ELSE
           BEGIN
              .           ! process the record (the returned
              .           ! <count—read> parameter tells
              .           ! the record length in bytes)
           END;
   END;          ! read loop
```

Example 3. Reading Sequentially by Alternate Key

Assume that you want to read only those records that contain the terminal number ATM37 in a particular data field that was defined during file creation as an alternatekey field. Assume also that the key specifier for that field is TN. You use KEYPOSITION to specify TN as the key specifier and ATM37 as the key value. You then execute a read loop. As a result of the KEYPOSITION call, Enscribe uses the alternate-key file associated with TN to access the desired data records from the primary file. The read loop terminates upon encountering the EOF mark in the alternate-key file.

```
STRING value [0:4];
INT specifier,
    compare^length;

error := FILE_OPEN_ (filename:length, filenum, ... );

specifier := "TN";
value ':=' "ATM37";
compare^length := 5;
generic := 1;
CALL KEYPOSITION (filenum, value, specifier,
                  compare^length, generic);

read^count := 200;
eof := 0;
```

```
    WHILE NOT eof DO
      BEGIN ! read loop
        CALL READ (filenum, buffer, read^count, count^read);
        IF > THEN eof := 1
        ELSE
           IF < THEN ... ! error
           ELSE
            BEGIN
                     .        ! process the record (the returned
                     .        ! <count—read> parameter specifies
                     .        ! the actual record length in bytes)
           END;
      END; ! read loop
```

Example 4. Reading Randomly by Unique Alternate Key

Assume that a particular entry-sequenced file contains transaction summary records and that
one of the defined alternate-key fields in each record contains a unique transaction number.
Assume also that the key specifier for that field is TX. If you want to read the data record for
transaction number AB0829, you use FILE_SETKEY_ , with exact positioning, to locate the
record and then use READ to read the record.

```
STRING value [0:5];
INT specifier,
    value^length;

error := FILE_OPEN_ (filename:length, filenum, ... );

specifier := "TX";
value ':=' "AB0829";
value^length := 6;
exact := 2;
error := FILE_SETKEY_ (filenum,
                        value:value^length,specifier,exact);
read^count := 200; ! maximum data record size
CALL READ (filenum, buffer, read^count, count^read);

! the returned <count—read> parameter
! specifies the actual record length in bytes
```

9 Relative Files

Enscribe Relative Files

Enscribe relative files consist of fixed-length physical records on disk that are accessed by relative record number. A record number is an ordinal value and corresponds directly to the record's position in the file. The first record is identified by record number zero; succeeding records are identified by ascending record numbers in increments of one.

Figure 20 (page 133) illustrates the structure of a relative file.

Each physical record position in a relative file occupies a fixed amount of space and each can contain one variable-length data record (logical record). A logical record can vary in size from zero, an empty record, to the maximum record size specified when the file was created. You can change a record's logical length after it has been written to the file, but the lengths of all logical records in the file must always be less than or equal to the constant size of the physical record. Each logical record has a length attribute that can be returned when a record is read. Logical records in a relative file can be logically deleted by issuing a write with a specified length of zero.

Once you have created the file and written a data record to it, all physical records preceding that record are also created and actually occupy space on the disk even though they contain no data. For example, if you create a relative file and then write a data record to record number 135, records 0 through 134 are also physically created on the disk at that time even though they have a logical record length of zero.

Note that this characteristic represents a limiting factor that could influence whether or not you use the relative file type.

Figure 20 Relative File Structure
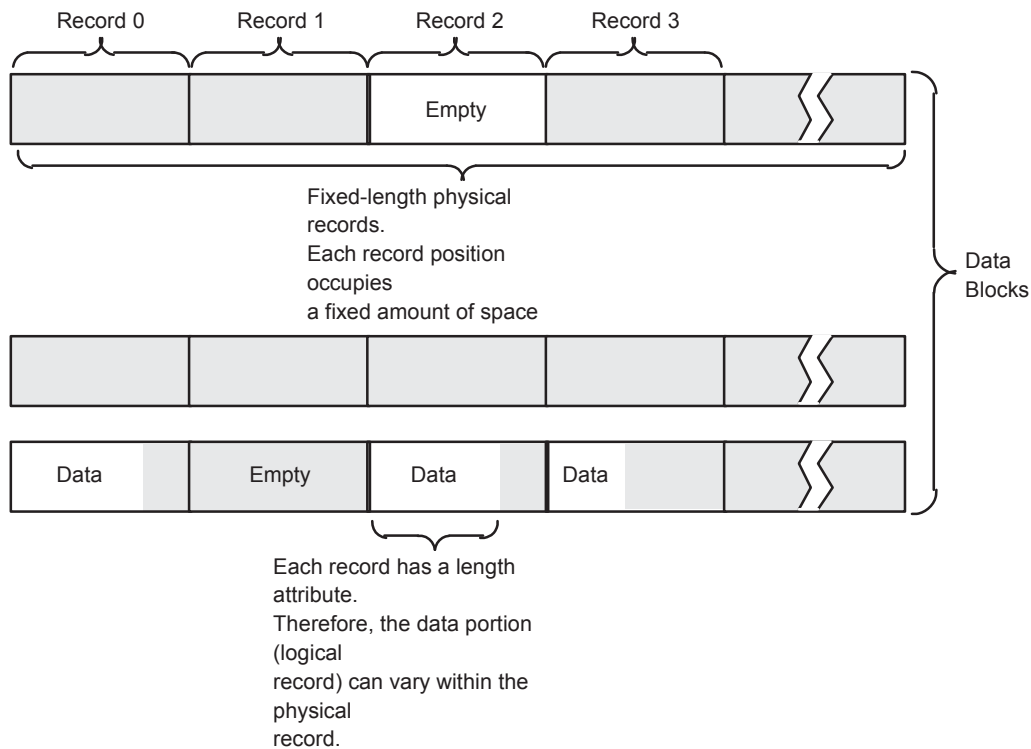
The exact position where a new record is to be inserted into a relative file is specified by supplying a record number to the FILE_SETPOSITION_ procedure. Alternatively, you can specify that

records be inserted into any available position by supplying a record number of -2 to FILE_SETPOSITION_ before inserting records into the file. You can specify that subsequent records be appended to the end of a file by supplying a record number of -1 to the FILE_SETPOSITION_ procedure.

For example, in a relative file in which only record number 10 contains data, 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 -2, 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 -2 or -1 is specified for inserting records into a relative file, the actual record number associated with the new record can be obtained through the FILE_GETINFOLIST_ procedure.

Relative files are best suited for applications where random access to fixed-length records is desired and where the record number has some meaningful relationship to a particular piece of data within each logical record. An inventory file, for example, could be a relative file with the part number serving as the record number. Such usage, however, would probably be very wasteful of disk space because part-numbering schemes often leave large gaps in the overall number sequence; this could result in many records being allocated but not used. An invoice file with the invoice number serving as the record number might be a better candidate for the relative file type because there are typically no large gaps in that type of numbering scheme. In the latter case, because your invoice numbers might begin at some large number such as 10000, you will likely have to use an address conversion algorithm to generate a record number sequence that begins at zero and then include the actual invoice number as a data field within the record.

Applicable System Procedures

- FILE_CREATE_, FILE_CREATELIST_

- FILE_OPEN_, FILE_CLOSE_, AWAITIO[X], FILE_AWAITIO64_

- FILE_LOCKFILE64_, FILE_LOCKREC64_, FILE_UNLOCKFILE64_, FILE_UNLOCKREC64_, LOCKFILE, LOCKREC, UNLOCKFILE, UNLOCKREC

- FILE_SETKEY_, FILE_SETPOSITION_, POSITION, KEYPOSITION, SAVEPOSITION, REPOSITION

- FILE_READ64_, FILE_READLOCK64_, FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READ[X], READLOCK[X], READUPDATE[X], READUPDATELOCK[X]

- FILE_WRITE64_, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITE[X], WRITEUPDATE[X], WRITEUPDATEUNLOCK[X]

- FILE_GETINFO_, FILE_GETINFOLIST_, FILE_GETINFOBYNAME_, FILE_GETINFOLISTBYNAME_

Types of Access

You can refer to specific records within a relative file either by their primary key (relative record number) or by the content of one or more alternate-key fields such as department number or zip code, for example, in an employee file.

There are three major pointers associated with an Enscribe relative file:

Current-record pointer	Specifies the physical record that was most recently read from or written to.
Next-record pointer	Specifies the next physical record that will be read from or written to.

| EOF pointer | Specifies the byte following the record with the highest address that currently contains data. Note that the same EOF pointer value is shared by all opens of a particular file. |

When the data records of a relative file include one or more alternate-key fields, there are two other values that are used for manipulating the pointers by way of alternatekey files:

| Current key specifier | Identifies which alternate-key field is to be used for accessing records in the file. When the current access path is by primary key, the current key specifier value is zero. |
| Current key value | Identifies the particular alternate-key value that is currently being used within the current access path. When the current access path is by primary key, the current key value is the address of the physical record that was most recently read from or written to. |

When you open a relative file, both the current-record and next-record pointers point to the first record in the file and the access path is by primary key.

You can change the access path from the primary-key field to an alternate-key field, from one alternate-key field to another, or back to the primary-key field at any time by using the FILE_SETKEY_, FILE_SETPOSITION_, KEYPOSITION, and POSITION system procedures. When the access path is by primary key, successive calls to the FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ, READLOCK, and WRITE system procedures access successively higher physical records in the file. You can change the content of the next-record pointer at any time to point to any specific record in the file using the FILE_SETPOSITION_ system procedure. You can also use FILE_SETPOSITION_ to change the content of the next-record pointer so that it points to the EOF position (for appending records to the end of the file) or to the next available empty record

When the access path is by a particular alternate key, successive calls to the FILE_READ64_, FILE_READLOCK64_, READ and READLOCK system procedures access successively higher logical records that contain a specified value (or partial value) in that field. You can change the current key-specifier and current key-value, and thereby the content of the next-record pointer, at any time to point to the first record in the file that contains a particular value (or partial value) in any alternate-key field using the FILE_SETKEY_ system procedure.

FILE_SETPOSITION_ always sets the access path to the primary key.

You can also change the access path from an alternate-key field back to the primary key FILE_SETPOSITION_ once you do so, the next-record pointer points again to the first record in the file (relative record number 0).

Creating Relative Files

You create Enscribe relative files by using the File Utility Program (FUP) or by calling either the FILE_CREATE_ procedure or the FILE_CREATELIST_ procedure.

When you create a partitioned (multivolume) file, the file system automatically creates all of the partitions of that file when the first partition is created.

If you are using a system procedure to create a relative file and the file contains alternate-key fields, you must also create one or more alternate-key files. If you are using FUP to create the primary-key file, however, FUP automatically creates any required alternate-key files.

When creating a relative file, you must consider the maximum logical record size, the data block length, and disk extent sizes.

Logical Records

A logical record is the unit of information transferred between an application program and the file system.

When creating a relative file, you must specify the maximum logical record length of that file. This parameter defines the size of the fixed-length physical records on the disk.

The maximum record size that you choose when creating a file depends upon the requirements of your application.

Format 1 Files

For relative format 1 files, the maximum length of a logical record is 24 bytes less than the block size. Using the maximum allowed block size of 4096, the absolute maximum logical record size allowed for relative files is 4072 bytes.

Format 2 Files

For relative format 2 files, the maximum length of a logical record is 48 bytes less than the block size. Using the maximum allowed block size of 4096, the absolute maximum logical record size allowed for relative files is 4048 bytes. The error 579 will be returned if the record size exceeds the applicable limit.

The data records that you write to a relative file can be of varying lengths, but none can exceed the maximum logical record size specified when the file was created. If you try to write a record that is longer than the defined maximum record length, the file system rejects the insert operation with an error 21 (illegal count).

Blocks

A block is the unit of information transferred between the disk process and the disk. A block consists of one or more logical records and, in the case of relative files, associated control information. This control information, which is used only by the system, is summarized in "Block Formats of Structured Files" (page 172).

The block size of an Enscribe relative file must be 512 bytes, 1 KB, 2 KB, or 4 KB.

The block size must include 22 or 44 bytes per block for block control information and 2 or 4 bytes per record for record control information. Therefore, the maximum number of records that you can store in each block is:

```
N = (block-size - 22) / (record-size + 2) for Format 1 Files
N = (block-size - 44) / (record-size + 4) for Format 2 Files
```

Regardless of the record length, the maximum number of records that can be stored in a single block is 511 for format 1 files and 32767 for format 2 files.

A record cannot span block boundaries (that is, it cannot begin in one block and end in another). Therefore, the block size for a relative file must be at least *record-length* + 2 + 22 bytes for format 1 files and *record-length* + 4 + 44 bytes for format 2 files

Disk Extent Size

When you create an Enscribe relative file, you can specify:

- The size of the primary and secondary extents. Format 1 files can have from 1 through 65,535 pages (where a page is 2048 bytes), while format 2 files can have from 1 through 536,870,912 pages

- The maximum number of extents to be allocated for the file (16 or more for a nonpartitioned relative file)

If you do not specify extent sizes, both the primary and secondary extents sizes default to one page.

If you do not specify a maximum number of extents, MAXEXTENTS defaults to 16.

For nonpartitioned relative files, you can change the MAXEXTENTS value dynamically during program execution using either a SETMODE 92 procedure call or a FUP ALTER command.

File Creation Examples

The examples that follow show how to create a relative file, a relative file with alternate keys, an alternate-key file, and a partitioned relative file.

Example 1: Creating a Relative File

This example shows how to create an employee data file in which the individual records are to be accessed by employee number. If the employee numbering scheme starts at zero or one (for the first employee) and proceeds sequentially upward in increments of one, it is reasonable to use a relative file.

Byte offset:

0	40	75	80	85
name	home-address	emp-#	dept-code	job-code

90	100	106	112
salary	hire-date	termination-date	
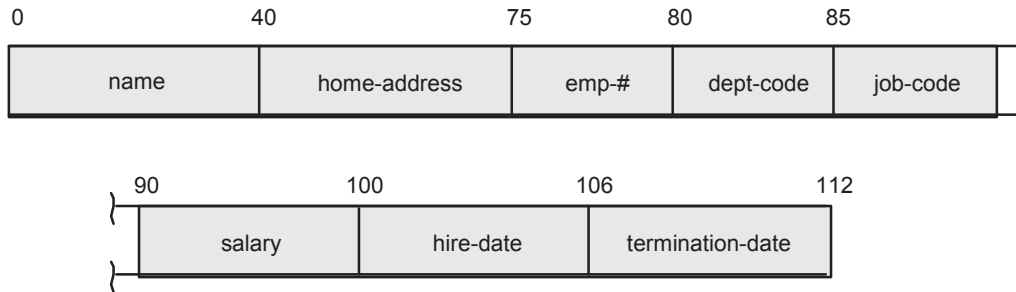
Note that while the employee number also happens to be included as data within the record, the records are actually accessed by their relative record number.

Assuming a format 1 file, with a record size of 112, selecting a block size of 4096 results in a blocking factor of 35 records per block:

```
N = (B - 22) / (R + 2)
35 = (4096 - 22) / (112 + 2)
```

If you designate the primary extent size as 60 pages and the secondary extent size as 30 pages, then the primary extent will accommodate 1050 employee records and each secondary extent will accommodate 525 additional employee records. When all 16 extents are eventually used, the file will accommodate a total of 8925 employee records.

You could create the file by using these FUP commands:

```
>volume $store1.svol1
>fup
-set type r
-set ext (60,30)
-set rec 112
-set block 4096
-show
    TYPE R
```

```
        EXT ( 60 PAGES, 30 PAGES )
        REC 112
        BLOCK 4096
        MAXEXTENTS 16
-create empfile
CREATED - $STORE1.SVOL1.EMPFILE
```

Using the FILE_CREATE_ system procedure, you could create the same file by including the TAL code in one of your application modules. The node name is not specified, so the FILE_CREATE_ procedure obtains the node name from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE. For more information on the =_DEFAULTS DEFINE, see the *TACL Programming Guide*.

```
LITERAL name^length = 21,
        pri^extent = 60,
        sec^extent = 30,
        file^type = 1,
        rec^len = 112,
        data^block^len = 4096;

INT namelen;
INT error;
STRING .filename [0:name^length-1] :=
        "$STORE1.SVOL1.EMPFILE";

namelen := name^length;

error := FILE_CREATE_ (filename:name^length,
        namelen,, pri^extent, sec^extent,,
        file^type,, rec^len, data^block^len);
IF error <> 0 THEN ... ! error
```

Example 2: Creating a Relative File With Alternate Keys

This example shows how to create the file illustrated in but defines the department code field as an alternate key.
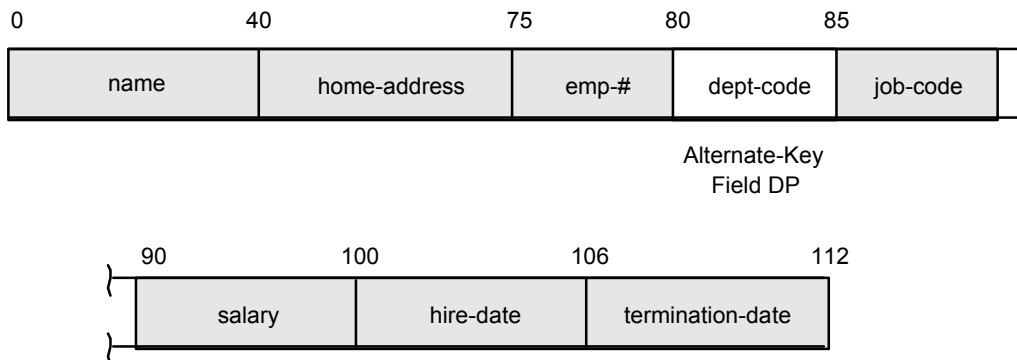
Byte Offset:

You could create the file by using these FUP commands:

```
>volume.$store1.svol1
>fup
-set type r
-set ext (60,30)
-set rec 112
-set block 4096
-set altkey ("DP",keyoff 80,keylen 5)
-set altfile (0,dept)
-show
        TYPE R
```

```
      EXT ( 60 PAGES, 30 PAGES )
      REC 112
      BLOCK 4096
      ALTKEY ( "DP", FILE 0, KEYOFF 80, KEYLEN 5 )
      ALTFILE ( 0, $STORE1.SVOL1.DEPT )
      ALTCREATE
-create empfile
CREATED - $STORE1.SVOL1.EMPFILE
CREATED - $STORE1.SVOL1.DEPT
```

Using the CREATE procedure, you could create the same file by including this TAL code in one
of your application modules. The node name and volume name for the new file are obtained from
the current value of the VOLUME attribute of the =_DEFAULTS DEFINE.

```
LITERAL name^length = 13,
        num^altkeys = 1,
        num^altkey^files = 1,
        item^list^len = 10;
INT error;
INT error2;
INT namelen;
STRING .filename [0:name^length-1] := "SVOL1.EMPFILE";

INT .item^list [0:item^list^len-1];

STRUCT value^list;
  BEGIN
    INT file^type;
    INT logical^reclen;
    INT block^length;
    INT pri^extent;
    INT sec^extent;
    INT altkeys;
    STRUCT altkey^descr [ 0:num^altkeys-1];
      BEGIN
        STRING key^specifier [0:1];
        INT key^length;
        INT key^offset;
        INT key^filenum;
        INT null^value;
        INT attributes;
      END;
```

```
            INT num^alt^key^files;
            STRUCT name^length^info [ 0:num^altkey^files-1];
              BEGIN
                INT file^name^len;
              END;
            STRING file^names [0:9];
          END;
?NOLIST
?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)
?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
?                                  FILE_OPEN_,
?                                  FILE_CREATELIST_,
?                                  READ)
?LIST
PROC DO^THE^WORK MAIN;
BEGIN
    namelen := name^length;

item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                ZSYS^VAL^FCREAT^LOGICALRECLEN,
                ZSYS^VAL^FCREAT^BLOCKLEN,
                ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                ZSYS^VAL^FCREAT^NUMALTKEYS,
                ZSYS^VAL^FCREAT^ALTKEYDESC,
                ZSYS^VAL^FCREAT^NUMALTKEYFILES,
                ZSYS^VAL^FCREAT^ALTFILELEN,
                ZSYS^VAL^FCREAT^ALTFILENAMES ];
                value^list.file^type := 1; ! relative
value^list.logical^reclen := 112;
value^list.block^length := 4096;
value^list.pri^extent := 60;
value^list.sec^extent := 30;
value^list.altkeys := num^altkeys;
value^list.altkey^descr[0].key^specifier ':=' "DP";
value^list.altkey^descr[0].key^length := 5;
value^list.altkey^descr[0].key^offset := 80;
value^list.altkey^descr[0].key^filenum := 0;
value^list.altkey^descr[0].null^value := 0;
value^list.altkey^descr[0].attributes := 0;
value^list.num^alt^key^files := num^altkey^files;
value^list.name^length^info[0].file^name^len := 10;
value^list.file^names ':=' "SVOL1.DEPT";

ERROR := FILE_CREATELIST_ (filename:name^length,namelen,
                           item^list, item^list^len, value^list,
                           $LEN(value^list), error2);
END;
```

Note that you must then create the alternate-key file separately. For more information see the following section.

Example 3: Creating an Alternate-Key File

When you use FUP to create the primary file, FUP automatically creates any required alternate-key files. If you create the primary file programmatically, however, you must create the alternate-key file yourself as a separate operation. This example assumes the use of a format 1 file. To see the difference with a format 2 file, see "Example 3: Creating a Key-Sequenced Format 2 File With Alternate Keys" (page 79).

You could create the alternate-key file for "Example 2: Creating a Relative File With Alternate Keys" (page 138) by including the TAL code in one of your application modules. Again, the node name is obtained from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE.

```
LITERAL name^length = 20,
        pri^extent = 30,
        sec^extent = 15,
        file^type = 3,
        rec^len = 11,
        data^block^len = 4096,
        key^length = 11,            ! max. alternate key length
                                    ! + primary-key length
                                    ! + 2
        key^offset = 0;

INT namelen;
INT error;
STRING .filename [0:name^length-1] :=
                "$STORE1.SVOL1.INVALT";

namelen := name^length;

error := FILE_CREATE_(filename:name^length,
                      namelen,, pri^extent, sec^extent,,
                      file^type,, rec^len, data^block^len,
                      key^length, key^offset);
IF error <> 0 THEN ... ! error
```

Example 4: Creating a Partitioned Relative File

This example shows how to create the file illustrated in Example 1 but enables it to ultimately span four partitions.

You could create the file by using these FUP commands:

```
volume $part1.svol1
>fup
-set type r
-set ext (60,30)
-set rec 112
-set block 4096
-set part (1,$part2,60,30)
-set part (2,$part3,60,30)
-set part (3,$part4,60,30)
-show
    TYPE R
    EXT ( 60 PAGES, 30 PAGES )
    REC 112
    BLOCK 4096
    PART ( 1, $PART2, 60, 30 )
    PART ( 2, $PART3, 60, 30 )
    PART ( 3, $PART4, 60, 30 )
-create empfile
CREATED - $PART1.SVOL1.EMPFILE
```

Note that each partition must reside on a separate disk volume. Within those volumes, however, the partitions all have the same subvolume name and file name (SVOL1.EMPFILE in this example). All four partitions are created at the same time.

When all 16 extents of the primary partition (#0) have been entirely used, the file system automatically begins using partition #1; when all 16 extents of that partition have been entirely used, the file system then begins using partition #2; and so forth.

Using the FILE_CREATELIST_ procedure, you could create the same file by including the TAL code in one of your application modules. The node name is not specified, so the FILE_CREATELIST_ procedure obtains the node name from the current value of the VOLUME attribute of the =_DEFAULTS DEFINE.

```
LITERAL name^length = 20,
        num^partitions = 3,
        item^list^len = 9;
```

```
      INT error;
      INT error2;
      INT namelen;
      STRING .filename [0:name^length-1] := "$PART1.SVOL1.EMPFILE";
      INT .item^list [0:item^list^len-1];

      STRUCT value^list;
        BEGIN
          INT file^type;
          INT logical^reclen;
          INT block^length;
          INT pri^extent;
          INT sec^extent;
          INT partitions;
          STRUCT part^info [0:num^partitions-1];
            BEGIN
              INT part^pri^extent;
              INT part^sec^extent;
            END;
          STRUCT vol^name^len [0:num^partitions-1];
            BEGIN
              INT vol^name^act^len;
            END;
          STRING vol^names [0:18];
        END;
      ?NOLIST
      ?SOURCE $SYSTEM.ZSYSDEFS.ZSYSTAL(FILESYSTEM^ITEMCODES)

      ?SOURCE $SYSTEM.SYSTEM.EXTDECS0(FILE_CLOSE_,
      ?                                FILE_OPEN_,
      ?                                FILE_CREATELIST_,
      ?                                READ)
      ?LIST
      PROC DO^THE^WORK MAIN;
      BEGIN
          namelen := name^length;
      item^list ':=' [ZSYS^VAL^FCREAT^FILETYPE,
                      ZSYS^VAL^FCREAT^LOGICALRECLEN,
                      ZSYS^VAL^FCREAT^BLOCKLEN,
                      ZSYS^VAL^FCREAT^PRIMEXTENTSIZE,
                      ZSYS^VAL^FCREAT^SCNDEXTENTSIZE,
                      ZSYS^VAL^FCREAT^NUMPRTNS,
                      ZSYS^VAL^FCREAT^PRTNDESC,
                      ZSYS^VAL^FCREAT^PRTNVOLLEN,
                      ZSYS^VAL^FCREAT^PRTNVOLNAMES];

      value^list.file^type := 1; ! relative
      value^list.logical^reclen := 112;
      value^list.block^length := 4096;
      value^list.pri^extent := 60;
      value^list.sec^extent := 30;
      value^list.partitions := 3;
      value^list.part^info[0].part^pri^extent := 60;
      value^list.part^info[0].part^sec^extent := 30;
      value^list.part^info[1].part^pri^extent := 60;
      value^list.part^info[1].part^sec^extent := 30;
      value^list.part^info[2].part^pri^extent := 60;
      value^list.part^info[2].part^sec^extent := 30;
      value^list.vol^name^len.vol^name^act^len[0] := 6;
      value^list.vol^name^len.vol^name^act^len[1] := 6;
      value^list.vol^name^len.vol^name^act^len[2] := 6;
      value^list.vol^names ':=' "$PART2$PART3$PART4";

      error := FILE_CREATELIST_ (filename:name^length,namelen,
```

```
item^list, item^list^len, value^list,
$LEN(value^list), error2);

END;
```

Accessing Relative File

The following paragraphs discuss the file pointers and access methods for Enscribe relative files.

The File Pointers

Separate next-record and current-record pointers are associated with each opening of a relative disk file so that if the same file is open several times simultaneously, each opening will provide 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 EOF pointer, however, is associated with all openings of a given relative disk file. This permits data to be appended to the end of a file by several different accessors. The EOF pointer resides in the file's file control block (FCB) in the disk-I/O process environment. A file's EOF pointer value is copied from the file label on disk when the file is opened and is not already open.

You can explicitly change the content of the next-record pointer to that of the EOF pointer by specifying an address of -1 in a POSITION or FILE_SETPOSITION_ call.

When appending to a file, the EOF pointer is advanced automatically each time a new block is added to the end of the data in the file. Note that in the case of partitioned files the EOF pointer relates only to the final partition containing data.

A file's EOF pointer is not automatically written through to the file label on disk each time it is modified. Instead, for relative files it is physically written to the disk only when one of these events occurs:

- A file label field is changed and the autorefresh option is enabled.

- The last accessor closes the file.

- The DISK_REFRESH_ procedure is called for the file.

- The REFRESH command is executed for the file's volume.

When using the FILE_CREATE_ procedure, you enable the autorefresh option by setting options bit 13 to 1. When using FUP, you enable the autorefresh option by issuing a SET REFRESH command.

For files created with the autorefresh option disabled, you can subsequently enable autorefresh by issuing a FUP ALTER command containing the REFRESH ON parameter

Effects of File-System Procedures on Pointers

The following paragraphs briefly describe how each applicable system procedure affects the current access path, current-record pointer, and next-record pointer.

`FILE_OPEN_`	Sets the current access path to the primary-key field.
	Changes the content of the current-record and next-record pointers so that they both point to record number zero.
`FILE_SETPOSITION,POSITION`	Sets the current access path to the primary-key field.
	Changes the content of the current-record and next-record pointers so that they both point to the particular record identified by the specified record number.
	Record number -1 resets both pointers to the current EOF position.

	Record number -2 resets both pointers to an available empty record in the file.
`FILE_SETKEY_`, KEYPOSITION	Sets the current access path to the specified key field. • With nonunique alternate keys: changes the content of the current-record and next-record pointers so that they both point to the first (lowest) record in the file that contains the specified key-value (or partial key-value) in the specified alternate-key field. • With unique alternate keys: changes the content of the current-record and next-record pointers so that they both point to the particular record in the file that contains the specified key-value in the specified alternate-key field.
`FILE_READ64_`, `FILE_READLOCK64_`, READ or READLOCK	Returns the content of the record pointed by the next-record pointer. When reading by primary key, FILE_READ64_, FILE_READLOCK64_, READ and READLOCK skip empty records. Upon completion, FILE_READ64_, FILE_READLOCK64_, READ and READLOCK change the pointers as: • current-record pointer = next-record pointer • next-record pointer = address of next higher record in current access path Note that if FILE_READ64_, FILE_READLOCK64_, READ or READLOCK skip an empty record, they increment the pointers one extra time for each record skipped so that the pointers maintain the proper values.
`FILE_WRITE64_` or WRITE	Writes to the record pointed to by the next-record pointer. The particular record must be empty; if not, the operation fails with an error 10 (record already exists). Upon completion, FILE_WRITE64_/WRITE changes the pointers as: • current-record pointer = next-record pointer • next-record pointer = address of next higher record in current access path

The following procedures do not alter the pointers:

`FILE_GETINFOLIST_`	Returns the values of the current-record pointer and the next-record pointer.
`FILE_READUPDATE64_`, `FILE_READUPDATELOCK64_`, READUPDATE or READUPDATELOCK	Returns the content of the record pointed to by the current-record pointer. The particular record must already contain data; if not, the operation fails with an error 11 (record not in file).
`FILE_WRITEUPDATE64_`, `FILE_WRITEUPDATEUNLOCK64_`, WRITEUPDATE or WRITEUPDATEUNLOCK	Writes to the record pointed to by the current-record pointer. The particular record must already contain data; if not, the operation fails with an error 11 (record not in file).

Sequential Access

The FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ, READLOCK, and WRITE system procedures provide sequential access to Enscribe relative files

If the current access path is by primary key, a succession of FILE_READ64_, FILE_READLOCK64_, READ or READLOCK calls obtains data from successively higher physical records in the file, skipping over empty physical records.

If the current access path is by a particular alternate-key field, a succession of FILE_READ64_, FILE_READLOCK64_, READ or READLOCK calls obtains data from successively higher physical records within the particular access path (that is, physical records that contain the current key-value in the alternate-key field identified by the current key-specifier).

An attempt to read beyond the last record in either access path returns an EOF indication.

When you are writing data records, a succession of FILE_WRITE64_/WRITE calls writes data to successively higher physical records in the file. Note that FILE_WRITE64_/WRITE will only write data to empty physical records. If you attempt to FILE_WRITE64_/WRITE to a physical record that already contains data, the operation fails with an error 10 (record already exists).

Upon completion of each FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ, READLOCK, or WRITE call, the current-record pointer is set to the present value of the next-record pointer (thereby pointing to the record that was just read or written) and the next-record pointer is incremented to point to the next higher physical record in the current access path.

When reading or writing sequentially by primary key, you can access different subsets of physical records by using the FILE_SETPOSITION_ or POSITION system procedure calls. After each FILE_SETPOSITION or POSITION call, the next FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ, READLOCK, or WRITE call accesses the physical record that you positioned to, and subsequent FILE_READ64_, FILE_READLOCK64_, FILE_WRITE64_, READ, READLOCK, or WRITE calls access successively higher physical records in the file.

When reading or writing sequentially by a particular alternate key, you can reposition to the start of a new alternate-key access path using the FILE_SETKEY_ system procedure. After each FILE_SETKEY_ call, the next FILE_READ64_, FILE_READLOCK64_, READ or READLOCK call retrieves the first (lowest) physical record in the file that contains the specified key value in the alternate-key field identified by the particular key specifier; subsequent FILE_READ64_, FILE_READLOCK64_, READ or READLOCK calls retrieve successively higher physical records in the particular access path.

Random Access

The update procedures FILE_READUPDATE64_, FILE_WRITEUPDATE64_, FILE_READUPDATELOCK64_, FILE_WRITEUPDATEUNLOCK64_, READUPDATE, WRITEUPDATE, READUPDATELOCK, and WRITEUPDATEUNLOCK, when used in conjunction with FILE_SETPOSITION_, provide random access to Enscribe relative files. The updating operation occurs at the record indicated by the current-record pointer. Random processing implies that a record to be updated must exist. If the physical record indicated by the current-record pointer is empty, the operation fails with an error 11 (record not found).

You cannot use FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE or WRITEUPDATEUNLOCK to alter a record's primary key. If you need to do so, you must first delete the record by using a FILE_WRITEUPDATE64_/WRITEUPDATE call with a write count of zero, and then insert the record into the desired physical record by using a FILE_SETPOSITION_ call followed by a FILE_WRITE64_/WRITE call.

If updating is attempted immediately after a call to FILE_SETKEY_ where a nonunique alternate key is specified, the operation fails with an error 46 (invalid key). However, if you issue an intermediate call to FILE_READ64_, FILE_READLOCK64_, READ or READLOCK, updating is then permitted.

Inserting Records

You insert data records into a relative file by using the FILE_WRITE64_/WRITE system procedure. The data record is written to the physical record indicated by the next-record pointer. Note that, for relative files, insertion requires that the referenced physical record be empty; if it is not, the operation fails with an error 10 (duplicate record).

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 fails with an error 10 (duplicate record).

Insertion of an empty record (one where `write-count` = 0) is not valid for relative files.

The length of a record to be inserted must be less than or equal to the record length defined for the file; if it is not, then the insertion fails with an error 21 (invalid count).

Deleting Records

Record deletion, which is accomplished by way of a FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE or WRITEUPDATEUNLOCK call with a `write-count` of zero, always applies to the physical record indicated by the current-record pointer.

File Access Examples

The remainder of this section presents annotated examples illustrating the most common ways to access Enscribe relative files.

Example 1 Sample Relative File

Most of the examples use the following sample file:

```
name               address          dept job salary region
                                     code code         code
                                     (DP)             (RG)
 0 nnnnnnnnnnnn aaaaaaaaaaaa    56   jjj ssssss 3
 1 (empty)
 2 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 4
 3 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 2
 4 nnnnnnnnnnnn aaaaaaaaaaaa    56   jjj ssssss 3
 5 nnnnnnnnnnnn aaaaaaaaaaaa    56   jjj ssssss 3
 6 nnnnnnnnnnnn aaaaaaaaaaaa    34   jjj ssssss 3
 7 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 4
 8 nnnnnnnnnnnn aaaaaaaaaaaa    34   jjj ssssss 3
 9 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 4
10 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 2
11 nnnnnnnnnnnn aaaaaaaaaaaa    56   jjj ssssss 6
12 (empty)
13 (empty)
14 nnnnnnnnnnnn aaaaaaaaaaaa    46   jjj ssssss 5
15 nnnnnnnnnnnn aaaaaaaaaaaa    46   jjj ssssss 1
16 (empty)
17 nnnnnnnnnnnn aaaaaaaaaaaa    60   jjj ssssss 1
18 nnnnnnnnnnnn aaaaaaaaaaaa    34   jjj ssssss 3
19 nnnnnnnnnnnn aaaaaaaaaaaa    46   jjj ssssss 4
```

Reading Sequentially by Primary Key

Using the sample file in the sequence of READ calls issued immediately after opening the file would read records 0, 2, 3, 4, and 5, respectively:

```
CALL READ (filenum, buffer, read^count);   ! reads record 0
CALL READ (filenum, buffer, read^count);   ! reads record 2
CALL READ (filenum, buffer, read^count);   ! reads record 3
```

```
CALL READ (filenum, buffer, read^count);   ! reads record 4
CALL READ (filenum, buffer, read^count);   ! reads record 5
```

Note that this sequence skips empty records (record 1 in the sample file).

If you want to start reading records sequentially by primary key from some specified point within the file, you could use a sequence such as:

```
CALL FILE_SETPOSITION_(filenum,10F); ! sets access
path = 0
                               ! (primary key) and the
                               ! next-record pointer = 10
CALL READ (filenum, buffer, read^count);   ! reads record 10
CALL READ (filenum, buffer, read^count);   ! reads record 11
CALL READ (filenum, buffer, read^count);   ! reads record 14
CALL READ (filenum, buffer, read^count);   ! reads record 15
CALL READ (filenum, buffer, read^count);   ! reads record 17
```

Note that the sequence skips empty records (records 12, 13, and 16 in the sample file).

Reading Sequentially by Alternate Key

This sequence defines the desired key value as 60, the desired key specifier as DP (department code), both the compare length and key length as 2 and exact positioning (2):

```
key^value := "60";
CALL FILE_SETKEY_(filenum, key^value:2,"DP",2);
```

Using those specifications and the sample file illustrated in "Sample Relative File" (page 146), a subsequent sequence of five READ calls accesses data records from the file as:

```
CALL READ (filenum, buffer, read^count);   ! reads record 2
CALL READ (filenum, buffer, read^count);   ! reads record 3
CALL READ (filenum, buffer, read^count);   ! reads record 7
CALL READ (filenum, buffer, read^count);   ! reads record 9
CALL READ (filenum, buffer, read^count);   ! reads record 10
```

Writing Sequentially by Primary Key

You write data to empty physical records in a relative file by using FILE_WRITE64_/WRITE procedure calls. The current access path must be the primary key field, which is the default when a relative file is first opened.

Assuming that records 0 through 4 are currently empty, this sequence of five WRITE calls issued immediately after opening the file would do this:

```
error := FILE_OPEN_ (filename:length, filenum)
                        ! sets the access path = 0
                        ! (primary key) and the
                        ! next-record pointer = 0
CALL WRITE (filenum, buffer, write^count); ! writes to rec. 0
CALL WRITE (filenum, buffer, write^count); ! writes to rec. 1
CALL WRITE (filenum, buffer, write^count); ! writes to rec. 2
CALL WRITE (filenum, buffer, write^count); ! writes to rec. 3
CALL WRITE (filenum, buffer, write^count); ! writes to rec. 4
```

If you attempt to WRITE data to a record that is not empty, the operation fails with an error 10 (duplicate record).

To append new records to the end of a relative file, you use the FILE_SETPOSITION_ procedure to point to the current EOF position and then issue successive WRITE calls. Assuming that the EOF pointer currently contains the value 260, this sequence of calls appends five records to the end of a relative file.

```
CALL FILE_SETPOSITION_(filenum,-1F);! sets next-rec.ptr.
= -1

CALL WRITE (filenum, buffer, write^count);!writes to rec. 260
CALL WRITE (filenum, buffer, write^count);!writes to rec. 261
```

```
CALL WRITE (filenum, buffer, write^count);!writes to rec. 262
CALL WRITE (filenum, buffer, write^count);!writes to rec. 263
CALL WRITE (filenum, buffer, write^count);!writes to rec. 264
```

Note that this sequence also causes the EOF pointer to be incremented appropriately.

After any of the write operations, you can obtain the address of the newly appended record by issuing this procedure call:

```
item := 202;
error := FILE_GETINFOLIST_ (filenum, item, 1, primary^key, 8);
```

To insert a new data record into any available empty physical record, you supply the value -2 in a FILE_SETPOSITION_ call and then issue a WRITE call.

```
CALL FILE_SETPOSITION_ (filenum, -2F);
CALL WRITE (filenum, buffer, write^count);
```

After doing so, you can obtain the address of the physical record that was actually used by issuing this procedure call:

```
item := 202;
error := FILE_GETINFOLIST_ (filenum, item, 1, primary^key, 8);
```

Random Access by Primary Key

You access individual records randomly within a relative file by using FILE_SETPOSITION_ calls to explicitly identify the desired record by primary key (relative record number).

```
CALL FILE_SETPOSITION_ (filenum,38F); ! sets next-rec.ptr=38
CALL WRITE (filenum, buffer, write^count); ! writes to rec 38
```

Note that this sequence assumes that record 38 is currently empty. If you attempt to WRITE data to a record that is not empty, the operation fails with an error 10 (file/record already exists).

```
CALL FILE_SETPOSITION_ (filenum, 76F); !sets next-rec.ptr=76
CALL READ (filenum, buffer, read^count); ! reads record 76
```

Note that this sequence assumes that record 76 actually contains data. If record 76 is empty, the READ call accesses the next record higher than 76 that does contain data.

Updating Records

To update a record, the particular physical record being accessed must already contain data. FILE_WRITE64_/WRITE cannot write to an occupied physical record. Conversely, FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE and WRITEUPDATEUNLOCK cannot write to an empty physical record.

After positioning to the particular record you wish to update, read the record using FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READUPDATE or READUPDATELOCK to verify that it contains data.

FILE_READ64_, FILE_READLOCK64_, READ and READLOCK always access the next higher nonempty record in the current access path, simply ignoring empty records. Consequently, they might or might not access the particular record that you positioned to. FILE_READUPDATE64_, FILE_READUPDATELOCK64_, READUPDATE and READUPDATELOCK, however, always access whatever record is pointed to by the current-record pointer. The use of these procedures (instead of FILE_READ64_, FILE_READLOCK64_, READ or READLOCK) therefore ensures that you are indeed examining the content of the particular record that you positioned to regardless of whether it contains data or is empty.

Having determined that the desired record contains data, you then need to change the desired data fields in your application buffer and write the revised record back to the disk using FILE_WRITEUPDATE64_, FILE_WRITEUPDATEUNLOCK64_, WRITEUPDATE or WRITEUPDATEUNLOCK.

Referring to the sample data file illustrated in "Sample Relative File" (page 146), this sequence of TAL statements updates the content of record 18:

```
CALL FILE_SETPOSITION_ (filenum, 18F); ! set next-record
                                        ! pointer = 18

CALL READUPDATE (filenum, buffer, read^count, count^read);

IF count^read > 0 THEN
  BEGIN
    buffer := changed^record;
    CALL WRITEUPDATE (filenum, buffer, write^count);
  END
ELSE ...        ! the specified record is empty;
                !    update not performed
```

10 File and Record Locking

Enscribe File and Record Locks

Access to shareable Enscribe files among two or more processes is coordinated by file locks and record locks.

Files can be either audited or nonaudited. An audited file is one that is being accessed under the control of TMF.

For nonaudited files, locks are granted on a file-open basis; that is, they are requested and granted in conjunction with individual file numbers associated with separate calls to the FILE_OPEN_ system procedure. A file or record lock through one file number prevents access to that file or record by any other file number, even within the same process.

For audited files, locks are granted on a transaction basis; that is, they are requested and granted in conjunction with individual transaction identifiers (transids) associated with separate BEGINTRANSACTION-ENDTRANSACTION statement pairs within an application program module. A file or record lock through one transid prevents access to that file or record through any other transid, even within the same process.

Multiple processes that are accessing the same disk file should always use either file or record locking before performing any critical sequence of operations to that file.

Locking Modes

Six locking modes are available; they are summarized in this table:

Table 11 Locking Modes

Normal mode	Any attempt to lock a file, or to read or lock a record that is already locked through a different file number or transid, is suspended until the existing lock is released. This is the default locking mode.
Reject mode	Any attempt to lock a file, or to read or lock a record that is already locked through a different file number or transid, is rejected with a file-system error 73 (file/record is locked); no data is returned.
Read-through Normal mode	FILE_READ64_, FILE_READUPDATE64_, READ and READUPDATE requests ignore existing record and file locks; encountering a lock does not delay or prevent reading the record. FILE_LOCKFILE64_, FILE_LOCKREC64_, LOCKFILE, LOCKREC, FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK, and READUPDATELOCK are treated as in normal mode.
Read-through/reject mode	FILE_READ64_, FILE_READUPDATE64_, READ and READUPDATE requests ignore existing record and file locks; encountering a lock does not delay or prevent reading the record. FILE_LOCKFILE64_, FILE_LOCKREC64_, LOCKFILE, LOCKREC, FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK, and READUPDATELOCK are treated as in reject mode.
Read-warn/normal mode	FILE_READ64_, FILE_READUPDATE64_, READ and READUPDATE requests ignore existing record and file locks; although an existing lock will not delay or prevent reading the record, it causes a CCG completion with a warning code of 9. FILE_LOCKFILE64_, FILE_LOCKREC64_, LOCKFILE, LOCKREC, FILE_READLOCK64_, FILE_READUPDATELOCK64_,

Table 11 Locking Modes *(continued)*

	READLOCK, and READUPDATELOCK are treated as in normal mode.
Read-warn/reject mode	FILE_READ64_, FILE_READUPDATE64_, READ and READUPDATE requests ignore existing record and file locks; although an existing lock will not delay or prevent reading the record, it causes a CCG completion with a warning code of 9. FILE_LOCKFILE64_, FILE_LOCKREC64_, LOCKFILE, LOCKREC, FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK, and READUPDATELOCK are treated as in reject mode.

You use SETMODE 4 procedure calls to enable the desired locking mode. The particular mode that you select determines what action occurs if you try to lock a file, or read or lock a record, when the file or record is already locked.

If you issue a CONTROL, FILE_CONTROL64_, FILE_WRITE64_ or WRITE request when the referenced file or any record within it is already locked through a different file number or transid, the request is always rejected immediately with an error 73 (file/record is locked) regardless of which locking mode you have selected.

File Locking

NOTE: A call to FILE_LOCKFILE64_/LOCKFILE is not equivalent to locking all of the records in the file; locking all records individually would still allow someone else to insert new records, whereas file locking would not.

You use FILE_LOCKFILE64_, FILE_UNLOCKFILE64_, LOCKFILE and UNLOCKFILE calls to lock and unlock Enscribe disk files.

If you call FILE_LOCKFILE64_/LOCKFILE when the specified file and all of the records within it are unlocked (or all of the existing record locks belong to you), the lock is granted and your application process continues execution.

If you call FILE_LOCKFILE64_/LOCKFILE when either the specified file or any record within it is already locked through a different file number or transid, the action taken depends upon the locking mode that is in effect at the time of the call.

When you close a file, all of the locks that you own against it are released immediately (except in the case of audited files, where the unlocking is delayed until all your transactions have been either committed or aborted and backed out).

NOTE: Frequent use of the FILE_LOCKFILE64_, FILE_UNLOCKFILE64_, LOCKFILE and UNLOCKFILE requests on partitioned files should be avoided, as there is additional overhead to lock and unlock every partition of the file. This overhead can be significant even for files with only 16 partitions. Hewlett Packard Enterprise recommends record level locking as a more efficient alternative with partitioned files.

Record Locking

Record locking operates in much the same manner as file locking, but it allows greater concurrent access to a single file.

These procedures lock records within a file:

- FILE_LOCKREC64_/LOCKREC locks the current record, as determined by the most recent operation against the file.

- FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK and READUPDATELOCK lock the record to be read before reading it.

- For audited files, FILE_WRITE64_/WRITE locks the record that is being inserted.

Note that if generic locking is enabled for a key-sequenced file, the FILE_LOCKREC64_/LOCKREC procedure also locks any other records in the file whose keys begin with the same character sequence as the key of the referenced record. Generic locking applies only to key-sequenced files. If both the referenced record and its file are unlocked (or the existing locks belong to you) when you call FILE_LOCKREC64_, FILE_READLOCK64_, FILE_READUPDATELOCK64_, LOCKREC, READLOCK, or READUPDATELOCK, the lock is granted and your application process continues execution.

If either the referenced record or its file is locked through a different file number or transid when you call FILE_LOCKREC64_, FILE_READLOCK64_, FILE_READUPDATELOCK64_, LOCKREC, READLOCK, or READUPDATELOCK, the action taken depends upon the locking mode that is in effect at the time of the call.

These procedures unlock records within a file:

- FILE_UNLOCKREC64_/UNLOCKREC unlocks the current record, as determined by the most recent operation against the file.

- FILE_UNLOCKFILE64_/UNLOCKFILE unlocks all of the records in the specified file that were locked through the file number or transid associated with the FILE_UNLOCKFILE64_/UNLOCKFILE call. In addition, if the file itself is locked through that same file number or transid, FILE_UNLOCKFILE64_/UNLOCKFILE releases the file lock as well.

- For nonaudited files, FILE_WRITEUPDATEUNLOCK64_/WRITEUPDATEUNLOCK releases the associated record lock after writing or deleting the record.

- For audited files, ENDTRANSACTION unlocks all of the records that the transaction had locked but does not do so until the transaction has been either committed or aborted and backed out.

- For nonaudited files, FILE_CLOSE_ unlocks all of the records that were locked through the file number associated with the call.

Note that if generic locking is enabled for a key-sequenced file, calls to the FILE_UNLOCKREC64_/UNLOCKREC procedure are ignored.

When you delete a record from a nonaudited file, the record lock is released immediately. When you delete a record from an audited file, however, the lock is not released until the transaction has been either committed or aborted and backed out.

Record locking provides maximum concurrency of access to a file while still guaranteeing the logical consistency of the file's content. However, for complex updating operations involving many records, record locking can involve a lot of system processing, consume a lot of memory required by the locks, and increase the possibility of deadlock. In such cases, file locking might be preferable.

Generic Locking

In addition to locking entire files or individual records within files, you can lock sets of records within key-sequenced files whose keys all begin with the same character sequence. This is referred to as generic locking.

You enable generic locking by issuing a SETMODE 123 procedure call that defines a generic lock key length that is shorter than the key length defined for the particular file. You disable

generic locking by issuing a SETMODE 123 procedure call that defines a generic lock key length that is either zero or equal to the key length defined for the particular file.

When generic locking is disabled, a record lock obtained with the FILE_UNLOCKREC64_/LOCKREC file-system procedure locks a single record by locking the specific key of that record.

When generic locking is enabled, a record lock obtained with the FILE_UNLOCKREC64_/LOCKREC procedure locks all of the records whose keys begin with the same byte string.

Note that when generic locking is enabled for a particular file, it applies to all opens of that file.

The length of the generic lock key length must be less than the length of the entire key. Note that there are now two key lengths: the overall key length defined at file creation time and the generic lock key length; each is defined on a per-file basis.

Consider this example:

File X is a key-sequenced file with a defined key length of 6. It contains records with these keys:

```
Aabcde
A1aabb
A2bbbb
A21ccc
A27def
B4dddd
B5abcd
C9dddd
```

If generic locking is enabled with a generic lock key length of 2, locking the record A2bbbb with a FILE_UNLOCKREC64_/LOCKREC procedure call locks these subset of records:

```
A2bbbb
A21ccc
A27def
```

Note that the records containing the keys A21ccc and A27def are also locked because the first two bytes of their record keys are identical to those of the locked record (A2bbbb). The records containing the keys Aabcde and A1aabb are not locked, however, because the first two bytes of their record keys do not exactly match those of the locked record.

The generic lock key length is stored in the file label and can be modified with SETMODE 123 procedure calls.

You can obtain the current generic lock key length programmatically by using the FILE_GETINFOLIST_ system procedure. You can also obtain the current generic lock key length interactively by using the File Utility Program (FUP) INFO command or change it interactively by using the FUP ALTER command.

Generic locking is activated whenever the generic lock key length is nonzero and less than the key length of the file. Note that the generic lock key length is initialized to the defined key length of the file so that generic locking is initially deactivated.

The lengths of all generic key locks possible at any one time on a file is a constant. This means that the generic lock key length cannot be changed if there are currently any locks in effect for the file.

When generic locking is activated, calls to the FILE_UNLOCKREC64_/UNLOCKREC procedure are ignored. Thus, with generic locking activated, you cannot unlock records either generically or individually. You can, however, unlock all records in the file with the FILE_UNLOCKFILE64_/UNLOCKFILE procedure.

The decision whether or not to use generic key locking should be made on a per-file basis and applies to all processes that lock records in that file. Therefore, you should carefully analyze all application programs that use that file to be certain that the benefits of generic locking to one application are not offset by drawbacks to another application.

The applications that benefit the most from generic locking are those that need to concurrently lock a set of records that together form a generic subset. One generic lock could potentially replace hundreds or thousands of individual record locks. In some application environments, the use of generic locking could both improve performance (because there will be fewer calls to the lock manager to allocate, deallocate and search locks) and reduce memory use (because there are fewer locks).

Applications that should not use generic locking are those that lock individual, unrelated records.

Note that if one process owns a generic lock on a subset of records and a second process tries to insert a record into that subset, the attempted insertion will fail and the second process will get an error 73 (file/record is locked) whether or not the record actually exists.

For example, assume that there exists an empty key-sequenced file with a defined key length of four and that generic locking is enabled with a generic lock key length of two. If process #1 inserts a record with the key AAaa, it thereby owns a generic lock for the byte-string AA. If process #2 at that time attempts to insert a record with the key AAcc, the attempt is rejected with a CCL and an error code 73.

Interaction Between File Locks and Record Locks

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

File locks take precedence over record locks. If you lock a file and then attempt to lock individual records within that file, the record lock requests are ignored and have no effect.

For each file having one or more pending file lock requests, the requests are queued. In addition, read calls that attempt to access a locked file are queued with the pending file lock requests. When the current lock is released, the system grants whatever request is currently at the head of the file lock queue. If the request is a file lock request, the lock is granted; if the request is a read request, the read operation is performed.

Similarly, for a record having one or more pending record lock requests, the requests are queued. In addition, read calls that attempt to access a locked record are queued with the pending record lock requests. When the current lock is released, the system grants whatever request is currently at the head of the queue for that record. If the request is a record lock request or a READUPDATELOCK/FILE_READUPDATELOCK64_ request, the lock is granted; if the request is a read request, the read operation is performed.

Lock requests do not wait behind other locks held by the same file number or transid. If a file number or transid holds record locks and later requests a file lock (and no other file number or transid has any locks pending against that file or any of its records), the record locks are relinquished and replaced by the file lock.

Lock Limits

The disk process enforces lock limits that are specified through SYSGEN. For nonaudited files, the default limit is 5000 locks per volume for each file number; for audited files, the default limit is 5000 locks of all kinds per volume for each transid. For partitioned files, these criteria translate into a default of 5000 locks per partition.

The limits do not imply that you can always get the maximum number of locks. For example, there might not be enough physical memory space or internal buffer space available to get another lock.

If the limit has been reached and you ask for an additional lock, the lock request is rejected with an error 35 (unable to obtain I/O process control block). The disk process returns different error codes for two other specific causes: error 37 (I/O process is unable to lock physical memory) when there is not enough physical memory available and error 33 (I/O process unable to obtain I/O segment space) when a buffer is full or too fragmented.

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

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:

Process A	Process B
`LOCKREC: record 1`	`LOCKREC: record 2`
.	.
.	.
`LOCKREC: record 2`	`LOCKREC: record 1`

Here, process A has record 1 locked and is requesting a lock for record 2, while process 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, this illustrated situation could never happen if each process requested the lock to record 1 before it requested the lock to record 2.

Because it is sometimes impossible for an application program to know in which order the records it must lock are going to be encountered, this approach is worth considering. For updates to single records of the file, no special processing needs to 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 processes requiring multiple records, because they must first gain access to the common record, but still allows maximum concurrency and minimum overhead for accessors of single records.

File Locking and Record Locking With Unstructured Files

You lock and unlock unstructured files in essentially the same way that you do structured files. You lock records in unstructured files by positioning the file to the relative byte address of the record to be locked and then calling the FILE_LOCKREC64_, LOCKREC, FILE_READLOCK64_, FILE_READUPDATELOCK64_, READLOCK, or READUPDATELOCK procedure. Any other user attempting to access the file at a point beginning at exactly that address sees the address as being locked; the action taken is governed by the current locking mode.

NOTE: Only the starting point specified by the relative byte address is locked. Another application could access part of the record if it specified a relative byte address somewhere else within the record.

TMF Locking Considerations

When your application is running under the control of TMF, every transaction must lock all of the records that it updates or deletes. To do so, the transactions can use either file locks or record locks.

When a process changes an audited database, the disk process imposes these constraints to prevent other transactions from either reading uncommitted changes or performing conflicting operations:

*   Whenever a transaction inserts a new record into an audited file, the disk process automatically obtains a lock based on the inserted record's primary-key value. This lock prevents any other transaction from inserting a record with the same primary-key value as

the newly inserted record, and from reading, locking, updating, or deleting the newly inserted record.

- Before a transaction can update or delete an existing record in an audited file, the transaction must previously have locked either the record or its file. If the transaction does not do so, the attempt to update or delete the record is rejected immediately with an error 79.

- The disk process retains all locks on inserted, updated, or deleted records in audited files until the associated transactions are either committed or aborted and backed out. Note, however, that for an FILE_UNLOCKREC64_/UNLOCKREC call following a FILE_READLOCK64_/READLOCK call, the lock is released immediately.

- Transactions should lock all of the records that they read and use in producing output regardless of whether the transaction modifies the data. Doing so guarantees that the data on which each transaction depends does not change before the transaction is either committed or aborted and backed out.

By performing heavy activity against an audited database, an application process can generate a large number of locks and thereby inadvertently create deadlock situations with other application processes. Therefore, when designing your application, you should consider the coordinated use of file locks and/or generic record locks among those processes that need access to the same database.

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

Figure 21 illustrates how processes can acquire locks and update audited files and when the disk process will release the locks.

If the complete set of currently active transactions requires too many locks, the attempt is rejected with an error 33, 35, or 37.

File locks and record locks are owned by the current transid of the 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 transid and the locks are held by the same transid, the locks do not cause conflict among the processes participating in the transaction.

Figure 21 Record Locking for TMF

Transaction A

LOCK record 1
Change record 1
UNLOCK record 1
·
·
LOCK record 2
Delete record
2

Audited
File

Locks for records 1
and 2 will be held until
transaction A commits or is
aborted and backed out.

Transaction B

LOCKFILE
Change record 1
·
·
Change record 2
UNLOCKFILE

Audited
File

The file lock will be held until
transaction B commits or is
aborted and backed out.

Transaction C

LOCK record 1
Change record 1
·
·
LOCK record 2
No change to record 2
·
·
UNLOCKFILE

Audited
File

Lock for record 1 will be held
until transaction C commits
or
is aborted and backed out.

Lock for record 2 is released
immediately by
UNLOCKFILE.

Figure 22 (page 158) illustrates these principles:

- The terminal control process (TCP) interprets BEGINTRANSACTION and obtains the transid before requesting database activity from the servers.

- The transid is transmitted to the servers as part of the request message, and any disk activity performed by the servers is associated with the particular transid.

- The particular transid owns the lock(s); all servers that are performing work for the same transid 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 are operating on behalf of the same transid.

Figure 22 Record Locking by Transid
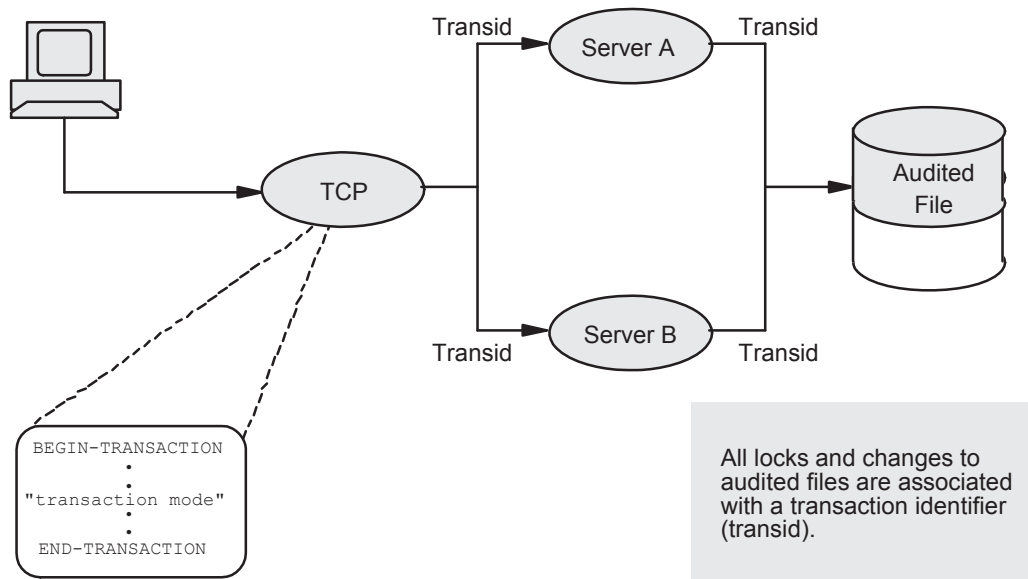
Errors in Opening Audited Files

With TMF, locks can persist longer than the opener process because the locks are owned by the transid instead of by the opener of a file. This means that even if a file has been closed by all its openers, the disk process will effectively keep it open until all transactions owning locks in the file have been either committed or aborted and backed out.

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

- Any attempt 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 that require exclusive access, such as PURGE and PURGEDATA, will fail. PURGE operations will fail with error 12 (file in use) while PURGEDATA operations will fail with error 80 (invalid operation on audited file).

Error 80 will also be returned if the OPEN call specifies unstructured access to a structured audited file.

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 a FILE_READ64_/READ call following exact positioning through the alternate file, the request fails with an error 1 (end-of-file) irrespective of the locking mode specified.

- If T2's request is a FILE_READ64_/READ call following exact positioning through the primary file, the request either fails with an error 73 (file or record is locked) if reject locking mode is in effect or it waits for T1 to complete if normal locking mode is in effect.

- If T2's request is a FILE_READUPDATE64_/READUPDATE call, the request either fails with an error 73 (file or record is locked) if reject locking mode is in effect or it waits for T1 to complete if normal locking mode is in effect. This behavior is true if the exact positioning is through the primary file. If the exact positioning is through the alternate file, the request waits for T1 to complete irrespective of 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 FILE_LOCKFILE64_/LOCKFILE procedure or carefully keep track of the number of locks held. If you do not use FILE_LOCKFILE64_/LOCKFILE, the disk process sets implicit locks as:

- When a new record is inserted into an audited file, the disk process implicitly locks that record.

- When a record is deleted from an audited file, the disk process implicitly retains a lock on that record.

These locks are not released until the transaction is either committed or aborted and backed out. This means that transactions doing batch updates to audited files can acquire too many locks if the transaction involves deleting, updating, or inserting a large number of records. The absolute maximum number of locks that can be acquired by each transaction is 5000. Any attempt to exceed the maximum number of locks will result in an error 33, 35, or 37.

If a TMF transaction calls FILE_LOCKFILE64_/LOCKFILE for a primary file, FILE_LOCKFILE64_/LOCKFILE is automatically applied to any associated alternate-key files. This prevents primary file updates from causing the alternate-key files to obtain record locks.

11 Errors and Error Recovery

Error Message Categories

The file system generates a number of messages indicating errors or other special conditions. You can encounter these messages during execution of any program module that uses system procedures.

Both a condition code and an error number are associated with the completion of each file-system procedure call. For successful completions, the condition code is CCE and the error number is 0.

The full range of error numbers can be subdivided into three general categories:

* Warnings issued by the file system (1 through 9).

* Errors encountered during standard file-system operations (10 through 255).

* Error numbers reserved for application-dependent use (300 through 511).

Many of the file-system error numbers indicate that you made a programming error, such as passing an invalid parameter or trying to initiate an illegal operation. Others indicate that the system is not being operated properly. Still others are essentially informational messages in that they merely inform you of particular device-oriented problems.

Communication Path Errors

A communication path error is a failure of a processor module, I/O channel, or disk controller port that is part of the primary path to disk unit. For errors of this type, the file system will attempt to switch to an alternate path and complete the I/O operation if you specified a sync depth greater than zero when opening the particular file.

An error number within the range 200 through 211 indicates that the operation is retryable. For specific information regarding communication path errors associated with queue files, see Chapter 7: Queue Files

Data Errors

A data error indicates that all or part of the file must be considered invalid.

Data errors are signified by the error numbers 50 through 59, 120 through 139, and 190 through 199.

Device Operation Error

Device operation errors are signified by the error numbers 60 through 69 and 103. The file system does not retry the failed operation when one of these types of error conditions occurs.

Errors 60-69 indicate that the device has been deliberately made inaccessible and, therefore, the associated operation should not be retried.

Error 103 indicates that the entire system has experienced a power failure and the disk is in the process of becoming ready. In such a case, you should periodically retry the failed operation.

Extent-Allocation Errors

Two error numbers are associated with programmatic allocation of disk extents: 43 (unable to obtain disk space for file extent) and 45 (file is full).

This example, in conjunction with Figure 23 (page 161), illustrates both types of error.

A file is created with an extent size of 2048 bytes. Repetitive WRITE operations of 400 bytes are then performed on the file:

```
loop: CALL WRITE ( filenum, buffer, 400, number^written );
      IF < THEN
```

```
            BEGIN
              status := FILE_GETINFO_ ( filenum, error );.
              ...
            END
        ELSE GOTO loop;
```

The first five WRITEs are successful (number^written = 400), but the sixth fails after transferring 48 bytes from the buffer to the disk (number^written = 48).

If insufficient disk space is available to allocate another extent, the error number returned by FILE_GETINFO_ is 43 (unable to obtain disk space for file extent).

If the condition occurred because the current extent is the last one permitted in the file, then the error number returned by FILE_GETINFO_ is 45 (file is full). You might be able to recover from an error 45 by using a SETMODE 92 call to dynamically increase the maximum number of extents.

If extents cannot be allocated because the file label is full, FILE_GETINFO_ returns error 43 (unable to obtain disk space for file extent). FILE_GETINFO_ does not return error 45 because, technically, the file is not full.

Note that an error 43 can also occur when allocating extents with a CONTROL 21 procedure call.

Figure 23 Example Showing Extent-Allocation Error
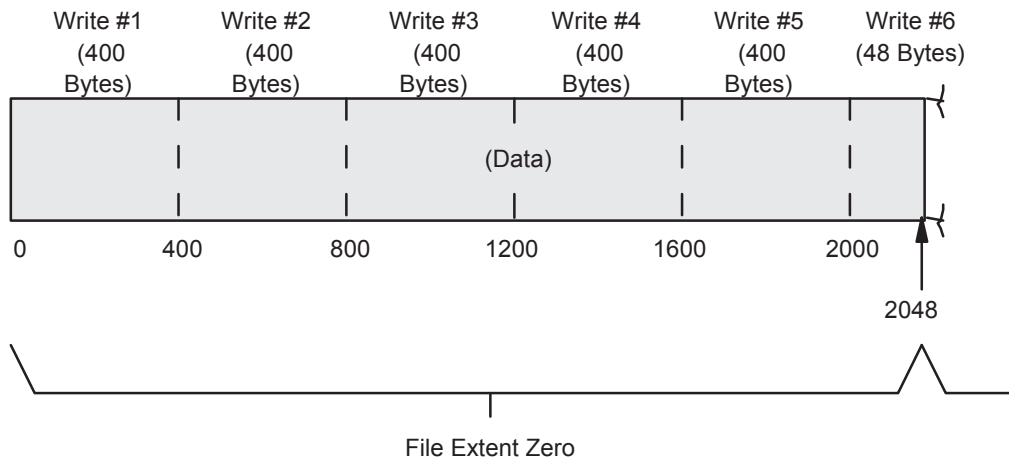
Errors and Partitioned Files

Each partition of a file can encounter error conditions independently of the file's other partitions. This is especially significant for errors 42 through 45, which pertain to disk space allocation. For errors 42 through 45 you might be able to correct the situation by using FUP to alter the size characteristics of the partition where the error occurred.

In any case, after each CCL or CCG return, you can obtain the particular error number by calling the FILE_GETINFO_ procedure, the partition number of the partition in error by calling the FILE_GETINFOLIST_ procedure, and the volume name of the partition in error by examining the file's partition parameter array either programmatically or by using FUP.

Failure of the Primary Application Process

A failure of the primary application process is actually 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 for the primary process. Refer to the discussion of checkpointing in the *Guardian Programmer's Guide* for information about how to recover from this type of error.

12 File Loading

File Utility Program (FUP) Commands

The File Utility Program (FUP) commands that you use to load data into an existing file are LOAD and LOADALTFILE.

The LOAD command loads data into an existing structured disk file without affecting any related alternate-key files. Any previous data in the file being loaded is lost. When loading data into key-sequenced files, the input records can be in either sorted or unsorted order; unless you specify the SORTED option, unsorted is assumed. For key-sequenced files, you can also specify slack space (a percentage of data block and index block space) to be left for future insertions.

The LOADALTFILE command generates alternate-key records from a specified primary-key file, sorts the generated records into ascending order by alternate-key value, and then loads the sorted records into a specified alternate-key file. You can specify slack space for future insertions. If your system does not have enough disk space available to perform the sorting operation in conjunction with the file loading operation, you can do the two operations separately. To do this, you would first use a FUP BUILDKEYRECORDS command to do the sorting; the sorted output can, for example, be directed to a magnetic tape. You would then use a FUP COPY or LOAD command to load the sorted records from the intermediate file into the destination alternate-key file.

The examples in this section illustrate file loading operations that require a sequence of FUP commands.

For more information about FUP, refer to the *File Utility Program (FUP) Reference Manual*.

Loading 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 up to but not including HA are to exist in the primary partition; records with primary-key values from HA up 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, with one record per block.

The FUP commands to perform this operation are:

```
-VOLUME $vol1.svol
-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.

Defining a New Alternate Key

This example defines a new alternate-key field for the primary file $VOL1.SVOL.PRIFILE, whose existing alternate-key file is $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 4, the key length is 20, and a null value of " " (blank) is specified for the new key field.

The FUP commands to perform this operation are:

```
-VOLUME $vol1.svol
-ALTER prifile, ALTKEY ( "NM", KEYOFF 4, KEYLEN 20, NULL " ")
-LOADALTFILE 0, prifile, ISLACK 10
```

The LOADALTFILE command loads PRIFILE's key file 0 ($VOL1.SVOL.ALTFILE) with the alternate-key records for key specifier NM and for any other alternate keys defined for key file zero. An index block slack percentage of 10 is specified.

Creating an Alternate-Key File

This example creates an alternate-key file for the primary file $VOL1.SVOL.FILEA, which is an entry-sequenced file. The new alternate-key file will be named $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 10.

The FUP commands to perform this operation are:

```
-VOLUME $vol1.svol
-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). Both 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 0 ($VOL1.SVOL.FILEB) with the alternate-key records for key specifier XY. An index block slack percentage of 0 is implied.

Reloading a Key-Sequenced File Partition

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 reloaded.

The FUP commands to perform this operation are:

```
-VOLUME $vol1.svol
-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 as a nonpartitioned 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 from the file $VOL1.SVOL.TEMP. The LOAD command includes the SORTED option because the records in the TEMP file are already in sorted order.

Creating a Partitioned Alternate-Key File

This example creates a partitioned form of an alternate key file. Partitioning allows a file to hold more data and to share I/Os between more than one physical disk. For this example, the primary file has a primary-key field 10 bytes long and an alternate key field 7 bytes long, with the key specifier "SN". The alternate key file is $VOL1.SVOL.ALTFILE, which has not been created yet.

The alternate key values starting with the letters "A" through "L" will have records placed into the first partition, and key values starting with the letters "M" through "Z" will be placed into the second partition. The partitions of the alternate key file will reside on volumes $VOL1 and $VOL2.

The FUP commands to perform this operation are:

```
-VOLUME $vol1.svol
-RESET
-SET TYPE K
-SET KEYLEN 19
-SET REC 19
-SET EXT (9000,1000)
-SET PART (1,$vol2,9000,1000,"SNM")
-CREATE altfile
```

If the primary file contains any data, the empty alternate key file will be loaded with the associated key records. In this example, the primary file is named PRIFILE, and it specifies ALTFILE as its alternate key file number 0.

The FUP command to load the newly created alternate file is:

```
-LOADALTFILE 0, prifile
```

For more information on FUP commands, see the *File Utility Program (FUP) Reference Manual*.

Loading a Partitioned, Alternate-Key File

For this example, primary file $VOL1.SVOL.PRIFILE is a key-sequenced file having a primary-key field 10 bytes long. The file has three alternate-key fields identified by the key specifiers F1, F2, and F3. Each of these alternate-key fields is 10 bytes long.

All of the alternate-key records are contained in a single 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 each alternate-key field is also the partial-key value for the associated secondary partition.

The alternate-key file's primary partition is $VOL1.SVOL.ALTFILE. That partition contains the alternate-key records for the key specifier F1. The first secondary partition, $VOL2.SVOL.ALTFILE, contains the alternate-key records for the key specifier F2. The second secondary partition, $VOL3.SVOL.ALTFILE, contains the alternate-key records for the key specifier F3.

The commands to load the alternate-key records for key specifier F2 into $VOL2.SVOL.ALTFILE are:

```
>FUP
-VOLUME $vol1.svol
-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.svol
-LOAD sortout, $vol2.altfile, SORTED, PARTOF $vol1, RECIN 22,
   BLOCKIN 2200
-PURGE ! sortin, sortout
```

The CREATE commands create the disk file used as the output of BUILDKEYRECORDS (which is also the input to SORT) and the disk 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 disk 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.

13 Support and other resources

Accessing Hewlett Packard Enterprise Support

- For live assistance, go to the Contact Hewlett Packard Enterprise Worldwide website:

    www.hpe.com/assistance

- To access documentation and support services, go to the HP Support Center – Hewlett Packard Enterprise website:

    www.hpe.com/support/hpesc

Information to collect

- Technical support registration number (if applicable)

- Product name, model or version, and serial number

- Operating system name and version

- Firmware version

- Error messages

- Product-specific reports and logs

- Add-on products or components

- Third-party products or components

Accessing updates

- Some software products provide a mechanism for accessing software updates through the product interface. Review your product documentation to identify the recommended software update method.

- To download product updates, go to either of the following:

    - HP Support Center – Hewlett Packard Enterprise **Get connected with updates from HP** page:

        www.hpe.com/support/e-updates

    - Software Depot website:

        www.hpe.com/support/softwaredepot

- To view and update your entitlements, and to link your contracts, Care Packs, and warranties with your profile, go to the HP Support Center – Hewlett Packard Enterprise **More Information on Access to HP Support Materials** page:

    www.hpe.com/support/AccessToSupportMaterials

ⓘ **IMPORTANT:** Access to some updates might require product entitlement when accessed through the HP Support Center – Hewlett Packard Enterprise. You must have a Hewlett Packard Enterprise Passport set up with relevant entitlements.

Websites

Website	Link
Hewlett Packard Enterprise Information Library	www.hpe.com/info/enterprise/docs
HP Support Center – Hewlett Packard Enterprise	www.hpe.com/support/hpesc
Contact Hewlett Packard Enterprise Worldwide	www.hpe.com/assistance
Subscription Service/Support Alerts	www.hpe.com/support/e-updates
Software Depot	www.hpe.com/support/softwaredepot
Customer Self Repair	www.hpe.com/support/selfrepair
Insight Remote Support	www.hpe.com/info/insightremotesupport/docs
Serviceguard Solutions for HP-UX	www.hpe.com/info/hpux-serviceguard-docs
Single Point of Connectivity Knowledge (SPOCK) Storage compatibility matrix	www.hpe.com/storage/spock
Storage white papers and analyst reports	www.hpe.com/storage/whitepapers

Customer self repair

Hewlett Packard Enterprise customer self repair (CSR) programs allow you to repair your product. If a CSR part needs to be replaced, it will be shipped directly to you so that you can install it at your convenience. Some parts do not qualify for CSR. Your Hewlett Packard Enterprise authorized service provider will determine whether a repair can be accomplished by CSR.

For more information about CSR, contact your local service provider or go to the CSR website:

www.hpe.com/support/selfrepair

Remote support

Remote support is available with supported devices as part of your warranty, Care Pack Service, or contractual support agreement. It provides intelligent event diagnosis, and automatic, secure submission of hardware event notifications to Hewlett Packard Enterprise, which will initiate a fast and accurate resolution based on your product's service level. Hewlett Packard Enterprise strongly recommends that you register your device for remote support.

For more information and device support details, go to the following website:

www.hpe.com/info/insightremotesupport/docs

Documentation feedback

Hewlett Packard Enterprise is committed to providing documentation that meets your needs. To help us improve the documentation, send any errors, suggestions, or comments to Documentation Feedback (**docsfeedback@hpe.com**). When submitting your feedback, include the document title, part number, edition, and publication date located on the front cover of the document. For online help content, include the product name, product version, help edition, and publication date located on the legal notices page.

A ASCII Character Set

Table 12 shows the USA Standard Code for Information Interchange (ASCII) character set, and the corresponding code values in octal notation.

Table 12 ASCII Character Set

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
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

Table 12 ASCII Character Set *(continued)*

!	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
@	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

Table 12 ASCII Character Set *(continued)*

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
h	064000	000150	Lowercase h
i	064400	000151	Lowercase i
j	065000	000152	Lowercase j

Table 12 ASCII Character Set *(continued)*

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

B Block Formats of Structured Files

This appendix describes the block formats for key-sequenced, queue, 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 relative files, an array of record lengths replaces the offsets map.

Figure 24 (page 173) shows the format 1 block and Figure 31 (page 178) shows the format 2 block.

The data area of a key-sequenced or queue 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.

The data area of a relative file also begins with a bit-map block telling which data blocks contain at least one record. The block immediately following a bit-map block is always the first data block.

For entry-sequenced files, all blocks in the data area are data blocks.

Figure 24 Block Format for Structured Format 1 Files

Decimal Offset	Field	Length in Bytes
[0]	eye-catcher	1
[1]	relative-sector-number	3
[4]	flags	1
[5]	index-level	1
[6]	volume-sequence-number	6
[12]	checksum	2
[14]	type-specific-block-header	[4, 6, 10, or 16]
[18, 20, 24, or 30]	(record 0)	
	(record 1)	
	⬚⬚⬚	
	(record N, where N = number-of-records -1)	
	free space (key-sequenced, queue, and entry-sequenced files only)	
	byte offset from [0] to start of free space / (reserved)	2
	byte offset from [0] to record N (key-sequenced, queue, and entry-sequenced files only) / size of record N (relative file only)	2
	⬚⬚⬚	
[blocksize - 2]	byte offset from [0] to record 0 (key-sequenced, queue, and entry-sequenced files only) / size of record 0 (relative file only)	2

Brackets: Common Block Header spans [0]–[12]; Data Area spans [18, 20, 24, or 30]; Offsets Map or Record-size Array spans the lower portion.

The fields in Figure 24 are defined as:

eye-catcher	is currently set to > but could 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 1 and 2: These two bits are reserved for decompression and are used internally for SQL compression logic.
	Bits 3 through 5: These three bits indicate the file type as:

000	(reserved)
001	Relative File

173

010	Entry-Sequenced File
011	Key-Sequenced or Queue File
100	(reserved)
101	(reserved)
110	(reserved)
111	Directory

Bits 6 and 7: These two bits indicate the block type as:

00	Data or Index
01	Bit Map (must be key-sequenced, queue, or relative file)
10	Free (must be key-sequenced or queue file)
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 disk. For an audited file, the volume sequence number is included in the auditcheckpoint (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. For a nonaudited file, the volume sequence number is included in the checkpoint AC record.
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. For more information on illustrations of the different block header types, see Figure 25 through Figure 29, Figure 30 illustrates the arrangement of bit-map blocks within key-sequenced, queue, and relative files.
record	can be a data or index record. The length of record N in a key-sequenced, queue, 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 record length for key-sequenced or queue files is the block size minus 34 (30 bytes for the header and 4 for the smallest possible offsets map).

The format of an index record is as:
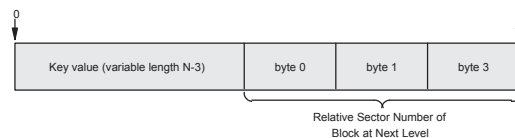
The *relative-sector-number* field in an index record points to the start of the block associated with this key. A null key value is used when KEYLEN = 0; this occurs in any index record pointing to an index block or to the first data record.

Figure 25 Index Block Header for Key-Sequenced and Queue Files

Offset (Decimal) / Length in Bytes

Offset	Field	Length
0	common-block-header	14
14	number-of-records-allocated	2
16	(reserved)	8
24	(record 0)	
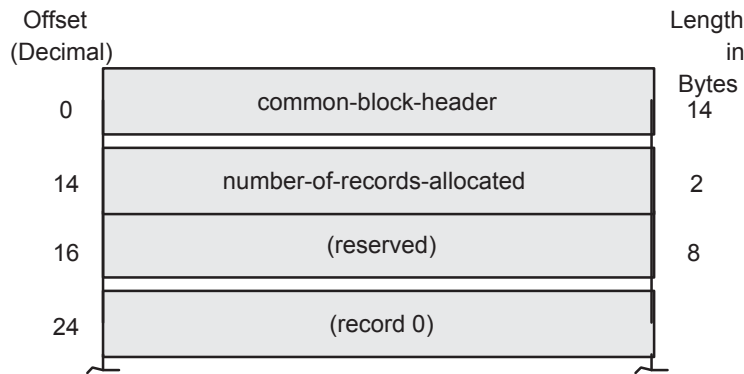
The fields in Figure 25 are defined as:

common-block-header is the 14-byte common block header shown in Figure 24.

number-of-records-allocated indicates how many records have been allocated in the block.

Figure 26 Data Block Header for Key-Sequenced and Queue Files

Offset (Decimal) / Length in Bytes

Offset	Field	Length
0	common-block-header	14
14	number-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)	
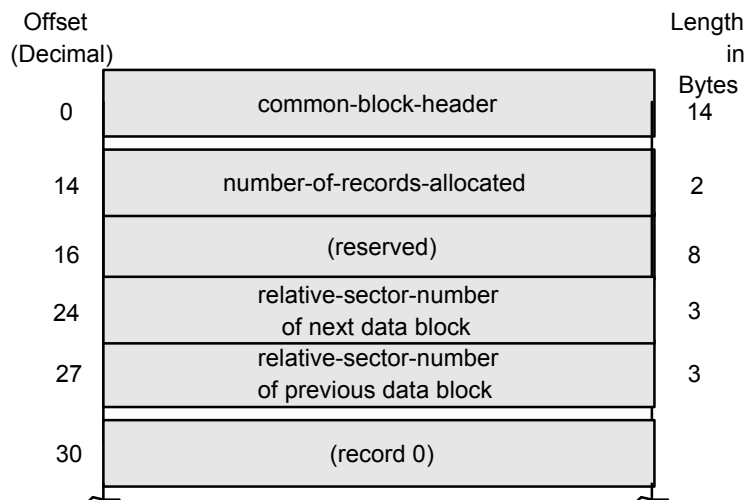
The fields in Figure 26 are defined as:

common-block-header is the 14-byte common block header shown in Figure 24.

number-of-records-allocated indicates how many records have been allocated in the block.

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.

relative-sector-number of previous data block provides a link to the previous logical block.

Figure 27 Header for Entry-Sequenced Data Block

Offset (Decimal)		Length in Bytes
0	common-block-header	14
14	number-of-records-allocated	2
16	(reserved)	4
20	(record 0)	
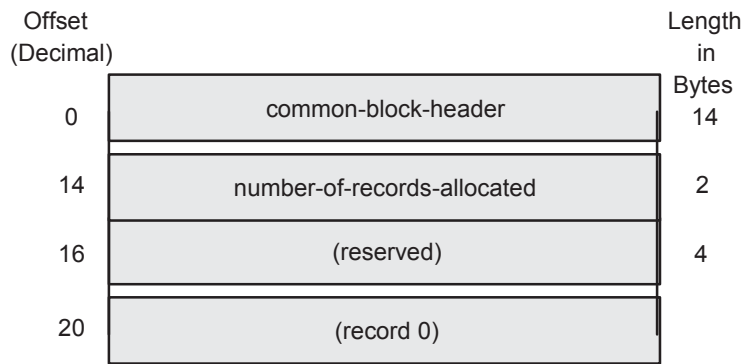
The fields in Figure 27 are defined as:

common-block-header is the 14-byte common block header shown in Figure 24.

number-of-records-allocated indicates how many records have been allocated in the block.

Figure 28 Header for Relative Data Block

Offset (Decimal)		Length in Bytes
0	common-block-header	14
14	number-of-records-allocated	2
16	number-of-records-present	2
18	(reserved)	2
20	(record 0)	
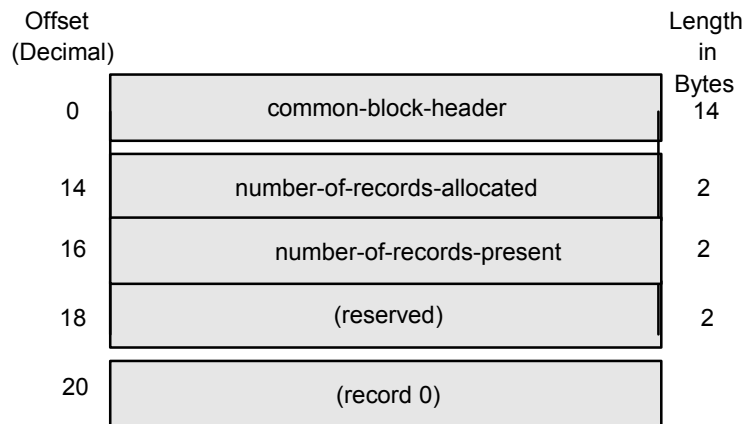
The fields in Figure 28 are defined as:

common-block-header is the 14-byte common block header shown in Figure 24.

number-of-records-allocated indicates how many records have been allocated in the block.

number-of-records-present indicates how many records are present in the block.

Figure 29 Header for Bit-Map Block

Offset (Decimal)		Length in Bytes
0	common-block-header	14
14	number-of-free-bits	4
18	bit-map	block size - 18
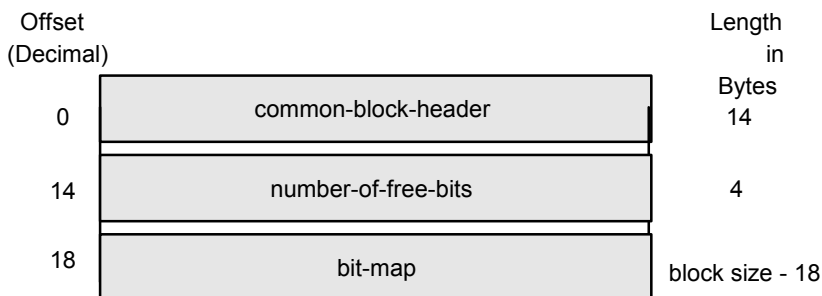
The fields in Figure 29 are defined as:

`common-block-header`	is the 14-byte common block header shown in Figure 24.
`number-of-free-bits`	indicates how many bits in this bit-map identify blocks that are free (empty) in a key-sequenced or queue file. For relative files, it indicates how many bits identify blocks that are not full.
`bit-map`	is an array of bits describing availability of index or data blocks. For a key-sequenced or queue 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.
	An empty bit-map has (8 * ($block\text{-}size$ - 18)) free bits.
	With a 1024-byte block, for example, the map has 8048 available bits.

Figure 30 Arrangement of Bit-Map Blocks
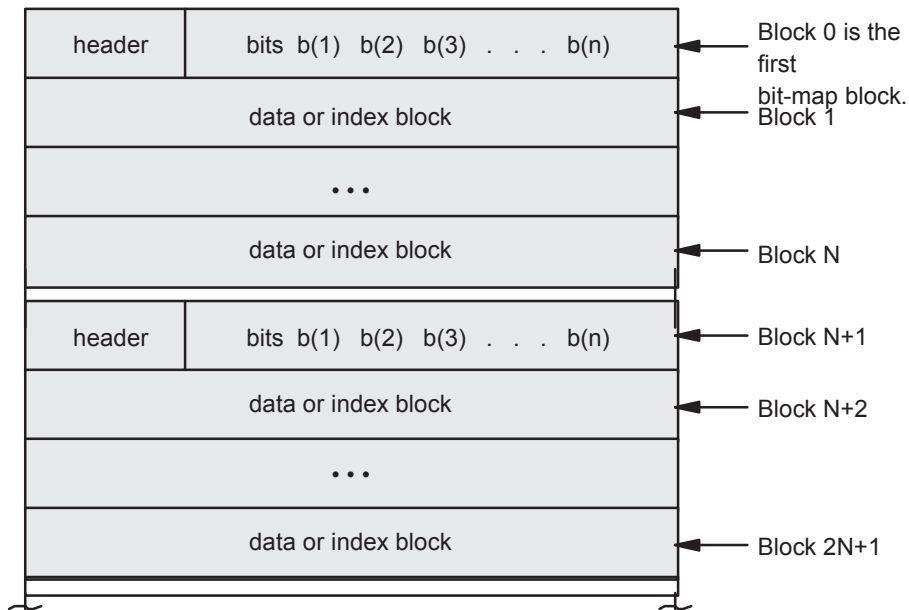
Figure 31 Block Format for Structured Format 2 Files
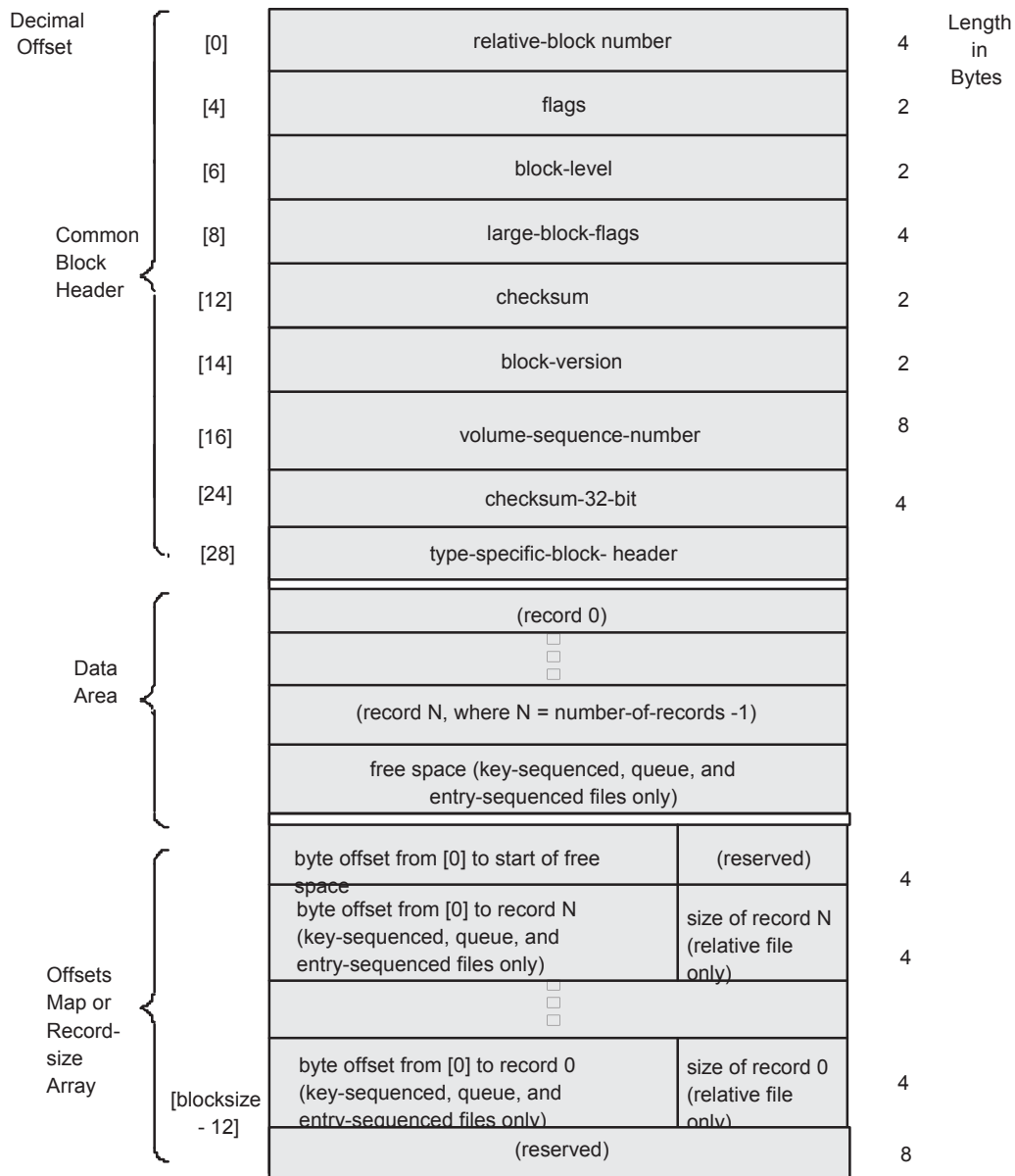
The fields in Figure 31 are defined as:

`relative-block-number`	identifies the relative block number within the file.
`flags`	Bit 0: This bit is set (=1) if the block is broken (inconsistent).
	Bits 1 and 2: These two bits are reserved for decompression and are used internally for SQL compression logic.
	Bits 3 through 5: These three bits indicate the file type as:

000	(reserved)
001	Relative File
010	Entry-Sequenced File
011	Key-Sequenced or Queue File
100	(reserved)
101	(reserved)

110	(reserved)
111	Directory

Bits 6 and7: These two bits indicate the block type as:

00	Data or Index
01	Bit Map (must be key-sequenced, queue, or relative file)
10	Free (must be key-sequenced or queue file)
11	(reserved)

block-level	contains the tree level of the block.
large-block-flags	Bit 0: This bit is set (=1) if the checksum is valid.
	Bit 1: This bit is set (=1) if the 32-bit checksum is valid.
	Bit 2: This bit is set (=1) if the partial checksum is used.
checksum	is the software checksum over the entire block.
block-version	identifies which format (1 or 2) is being used for the block
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 disk. For an audited file, the volume sequence number is included in the auditcheckpoint (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. For a nonaudited file, the volume sequence number is included in the checkpoint AC record.
checksum-32-bit	is a field reserved for future use and is currently unused.
record	can be a data or index record. The length of record N in a key-sequenced, queue, 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 record length for key-sequenced or queue files is the block size minus 56 (40 bytes for the header, 8 for the trailer and 8 for the smallest possible offsets map).

The format of an index record is as:
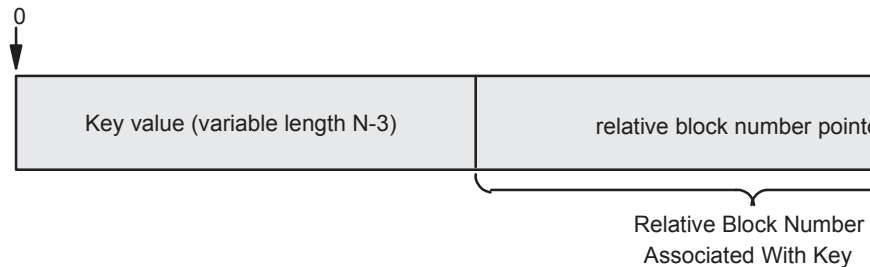
The *relative-block-number* field in an index record points to the start of the block associated with this key. A null key value is used when KEYLEN = 0; this occurs for the first record in an index block.

Figure 32 Index Block Header for Key-Sequenced and Queue Format 2 Files

```
Offset                                                    Length
(Decimal)                                                   in
                                                          Bytes
   0    |          common-block-header          |          28

  28    |       number-of-records-allocated     |           4

  32    |              (reserved)               |           8

  40    |              (record 0)               |
```
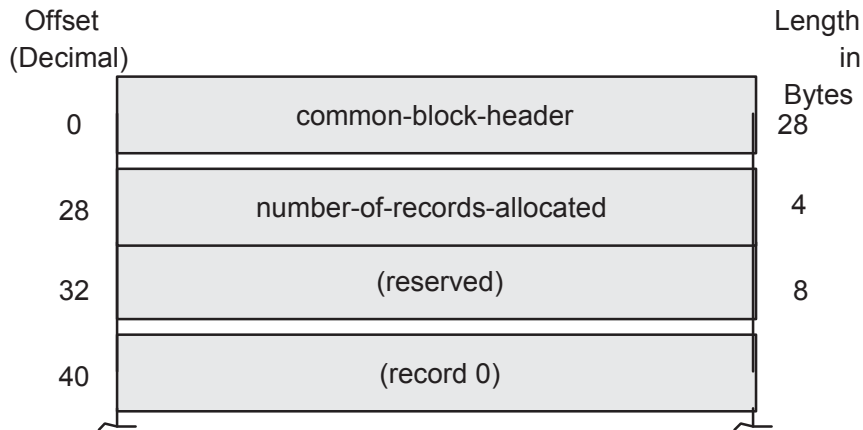
The fields in Figure 32 are defined as:

common-block-header   is the 28-byte common block header shown in Figure 31.

number-of-records-allocated indicates how many records have been allocated in the block.

Figure 33 Data Block Header for Key-Sequenced and Queue Format 2 Files

```
Offset                                                    Length
(Decimal)                                                   in
                                                          Bytes
   0    |          common-block-header          |          28

  28    |       number-of-records-allocated     |           4

  32    |         relative-block-number         |           4
        |          of next data block           |

  36    |         relative-block-number         |           4
        |        of previous data block         |

  40    |              (record 0)               |
```
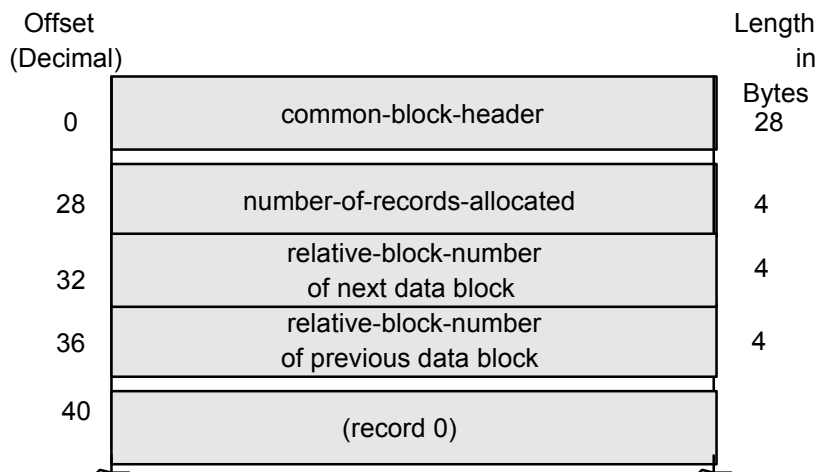
The fields in Figure 33 are defined as:

common-block-header   is the 28-byte common block header shown in Figure 31.

number-of-records-allocated indicates how many records have been allocated in the block.

relative-block-number of next data block   provides a link to the next logical block. The current block's *relative-block-number* is given in the common block header.

relative-block-number of previous data block   provides a link to the previous logical block.

Figure 34 Header for Format 2 Entry-Sequenced Data Block

Offset (Decimal)		Length in Bytes
0	common-block-header	28
28	number-of-records-allocated	4
32	(record 0)	2

The fields in Figure 34 are defined as:

common-block-header   is the 28-byte common block header shown in Figure 31.

number-of-records-allocated   indicates how many records have been allocated in the block.

Figure 35 Header for Format 2 Relative Data Block

Offset (Decimal)		Length in Bytes
0	common-block-header	28
28	number-of-records-allocated	4
32	number-of-records-present	4
36	(record 0)	4

The fields in Figure 35 are defined as:

common-block-header   is the 28-byte common block header shown in Figure 31.

number-of-records-allocated   indicates how many records have been allocated in the block.

number-of-records-present   indicates how many records are present in the block.

Figure 36 Header for Bit-Map Block

Offset (Decimal)		Length in Bytes
0	common-block-header	28
28	number-of-free-bits	4
32	bit-map	block size - 32

The fields in Figure 36 are defined as:

common-block-header	is the 28-byte common block header shown in Figure 31.
number-of-free-bits	indicates how many bits in this bit-map identify blocks that are free (empty) in a key-sequenced or queue file. For relative files, it indicates how many bits identify blocks that are not full.
bit-map	is an array of bits describing availability of index or data blocks. For a key-sequenced or queue 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.
	An empty bit-map has 8 * (block-size - 32) free bits.
	With a 1024-byte block, for example, the map has 7872 available bits.

C Action of Current Key, Key Specifier, and Key Length

This appendix contains pseudocode descriptions of the behavior of basic file-system operations and their relationships to file-currency information. The variables and functions used in the pseudocode are defined first. By evaluating the pseudocode, you can determine the action of the current key, key specifier, and key length for the different file-system operations.

Variable Definitions

These variables are used in the pseudocode descriptions:

CKV	*current key value*
CKS	*current key specifier*
CKL	*current key length*
CMPL	*comparison length*
MODE	*positioning mode* : (approximate = 0, generic = 1, exact = 2)
REVERSE	indicates whether the descending direction is to be used
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 (key-sequenced and queue files)
entryseq	file type 2
relative	file type 1

Function Definitions

These functions are used in the pseudocode descriptions:

* 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 number or record address 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, direction)

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

   ○ 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, direction)

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

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

Pseudocode Descriptions

These sets of pseudocode describe the behavior of the OPEN, KEYPOSITION, POSITION, READ, READUPDATE, WRITEUPDATE, and WRITE procedures.

OPEN (FILE_OPEN_)

```
CKS := primary
if keyseq then CKL := CMPL := 0
else
  begin
    if format I file format CKL := 4

    else CKL := 8;
    CKV := rip := 0;
  end;
MODE := approx;
next := false;
REVERSE := false;
```

FILE_SETKEY_, KEYPOSITION:

```
CKV := rip := key
if position-to-last then pad out CKV with %hFF
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;
REVERSE := reverse;
```

FILE_SETPOSITION_, POSITION:

```
CKV := rip := record specifier;
CKS := primary
if format I file format then CMPL := CKL := 4

  else CMPL := CKL := 8;
MODE := approx;
next := false;
REVERSE := false;
```

READ:

```
position := if next then find^next (MODE, CKS, CKV, CMPL,
                                    REVERSE)
```

```
                  else find (MODE, CKS, CKV, CMPL, REVERSE);
        if error then return;
        record := file[position];
        CKV := rip := keyfield (record, CKS);
        CKL := keylength (record, CKS);
        next := true;
```

READUPDATE:

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

WRITEUPDATE:

```
        position := find (exact, CKS, CKV, CKL, REVERSE);
        if error = 1 then error := 11; if error then return;
        if write count = 0 then
          if entryseq then begin error := 21; 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 := 46; return; end;
           if rip <> -2 and rip <> -1 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 := 46; return; end;
           position := insert (-1,4); ! end-of-file
           file[position] := record;
           CKV := keyfield (record, primary);
           next := true;
          end;
```

Index