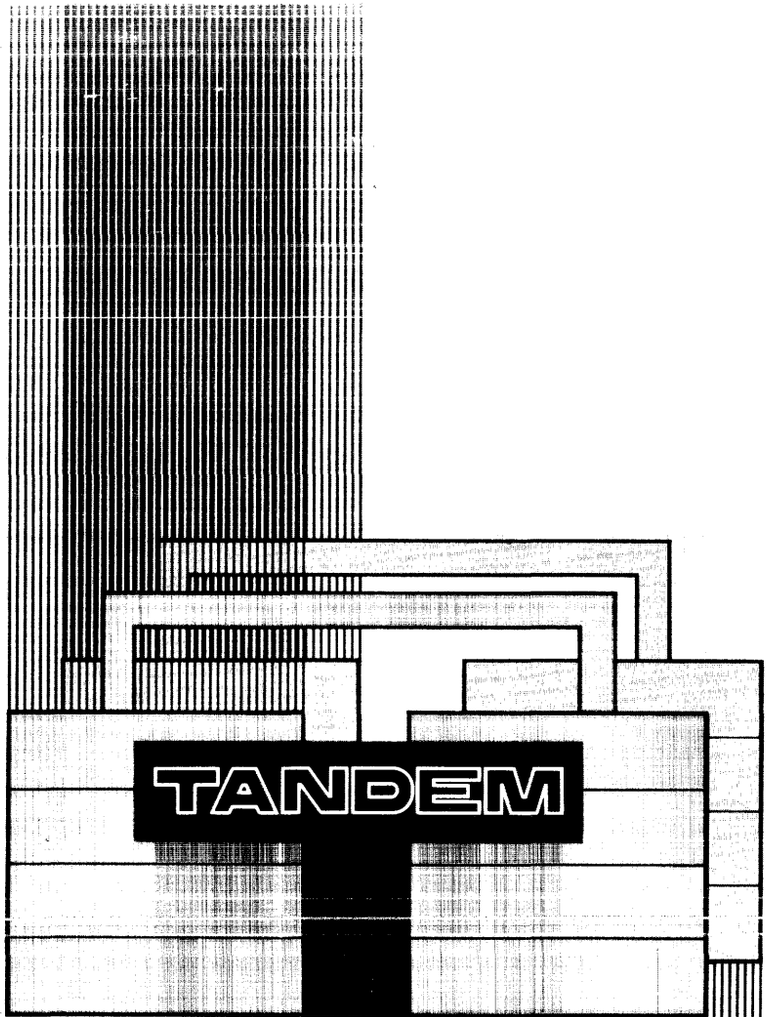
ENSCRIBE Programming Manual



ENSCRIBE Programming Manual

Tandem NonStop (TM) Systems

ENSCRIBE (TM) PROGRAMMING MANUAL

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, California 95014

P/N 82083

April, 1981
Printed in U.S.A.

PREFACE

This manual documents ENSCRIBE, Tandem's data base record manager. It is written for data base programmers and data base administrators whose job is to design, develop, and maintain data base applications for a Tandem NonStop System or a Tandem NonStop II System.

The manual contains both reference and background information. The primary source of reference material is Section 3, File Management Procedures, and Appendix E, File Management Procedure Syntax Summary. Section 3 contains detailed syntax descriptions of each of the ENSCRIBE file management procedures; the syntax descriptions are listed in alphabetical order.

Section 1, Introduction to ENSCRIBE, lays the foundation for all of the detailed information that follows. The introduction should be read at least once.

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

Section 3, File Management Procedures, contains syntax descriptions of all the ENSCRIBE calls.

Section 4, File Access, describes how ENSCRIBE files are accessed.

Section 5, File Creation, describes how to create ENSCRIBE disc files.

Section 6, File Loading, describes how to load data into ENSCRIBE files.

Appendix A, Sequential I/O Procedures, describes a set of procedures that can be used to perform common input and output operations on ENSCRIBE disc files.

An index is included for quick access to the mass of details that make up this book.

ENSCRIBE PROGRAMMING MANUAL

Table of Contents

SECTION 1. INTRODUCTION TO ENSCRIBE 1-1
Disc File Organization 1-2
Structured Files 1-3
Key-Sequenced File Structure 1-4
Relative File Structure 1-5
Entry-Sequenced File Structure 1-6
Multi-Key Access To Structured Files 1-7
Relational Access Among Structured Files 1-11
Automatic Maintenance of All Keys 1-12
Data and Index Compression 1-12
Unstructured Files 1-13
Access Coordination Among Multiple Accessors 1-14
Locking 1-15
File Locking 1-15
Record Locking 1-15
Wait/No-Wait I/O 1-16
Cache 1-20
Sequential Access Buffering 1-20
Multiple Volume (Partitioned) Files 1-21
File Creation 1-21
Data Definition Language 1-22
File Loading 1-22
Record Management Functions 1-23
File System Implementation 1-24
File and I/O System Structure 1-24
Hardware Structure 1-24
Software Structure 1-26
File System Procedure Execution 1-29
File Open 1-30
File Transfers 1-33
Buffering 1-35
File Close 1-37
Automatic Path Error Recovery for Disc Files 1-37
Mirror Volumes 1-44

SECTION 2. ENSCRIBE DISC FILES 2-1
Structured Files 2-1
Key-Sequenced Files 2-2
Relative Files 2-5
Entry-Sequenced Files 2-7
Accessing Structured Files -- Concepts 2-8
File 2-8
Record 2-8
Key 2-8
Primary Key 2-8
Alternate Key 2-8
Current Key Specifier and Curren Access Path 2-9
Current Key Value and Current Position 2-10
Positioning Mode and Compare Length 2-11
Approximate 2-11
Generic 2-11
Exact 2-11
Subset 2-12
Alternate Keys 2-12
Alternate Key Attributes 2-14
Alternate Keys in Key-Sequenced Files 2-14
Alternate Keys in Relative Files 2-14
Alternate Keys in Entry-Sequenced Files 2-15
Comparison of Structured File Characteristics 2-15
Unstructured Files 2-16
Characteristics 2-16
Relative Byte Addressing and File Pointers 2-18

SECTION 3. FILE MANAGEMENT PROCEDURES 3-1
File Management Call Summary 3-1
Characteristics of ENSCRIBE Calls 3-4
Completion 3-4
<file number> Parameters 3-5
<tag> Parameters 3-5
<buffer> Parameter 3-5
<transfer count> Parameter 3-6
Condition Codes 3-6
Errors 3-7
Access Mode and Security Checking 3-8
Current State Indicators 3-8
External Declarations 3-9
File Names 3-10
Permanent Disc File Names 3-11
Temporary File Names 3-11
File Name Examples 3-12
Network File Names 3-12

AWAITIO Procedure 3-14
CANCEL Procedure 3-21
CANCELREQ Procedure 3-22
CLOSE Procedure 3-23
CONTROL Procedure 3-24
CREATE Procedure 3-27
DEVICEINFO Procedure 3-39
EDITREAD Procedure 3-42
EDITREADINIT Procedure 3-46
FILEERROR Procedure 3-48
FILEINFO Procedure 3-51
FILERECINFO Procedure 3-56
FNAMECOLLAPSE Procedure 3-59
FNAMECOMPARE Procedure 3-61
FNAMEEXPAND Procedure 3-65
GETDEVNAME Procedure 3-70
KEYPOSITION Procedure 3-72
LOCKFILE Procedure (file locking) 3-78
LOCKREC Procedure (record locking) 3-82
NEXTFILENAME Procedure 3-86
OPEN Procedure 3-88
POSITION Procedure (relative, entry-seq, & unst files) ...3-98
PURGE Procedure 3-101
READ Procedure (sequential processing) 3-103
READLOCK Procedure (sequential processing, record locking) 3-108
READUPDATE (random processing).....3-111
READUPDATELOCK (random processing, record locking) 3-115
REFRESH Procedure 3-117
RENAME Procedure 3-118
REPOSITION Procedure 3-120
SAVEPOSITION Procedure 3-121
SETMODE Procedure 3-123
SETMODENOWAIT Procedure 3-125
UNLOCKFILE (file locking) 3-129
UNLOCKREC (record locking) 3-130
WRITE Procedure (insert) 3-132
WRITEUPDATE (random replace and delete) 3-136
WRITEUPDATEUNLOCK (random processing, record locking) 3-140

SECTION 4. FILE ACCESS 4-1
File Open 4-2
Access Rules for Structured Files 4-2
Sequential Processing 4-2
Random Processing 4-3
Inserting Records 4-3
Deleting Records 4-4
Alternate Keys 4-4
Current Position 4-4
Current Key Value 4-4
Current Primary Key Value 4-5
End-of-File Pointers 4-5
Sequential Buffering Option 4-6
Access Rules for Unstructured Files 4-7
Relative Byte Addressing and File Pointers 4-9
Sequential Access 4-12
Encountering EOF During Sequential Reading 4-13
Random Access 4-16
Appending to End-of-File....4-16
Disc Sectors 4-18
Resident Buffering (TNS only) 4-20
Considerations for Both Structured and Unstructured Files .4-23
Locking--General Concept 4-23
File Locking 4-23
Record Locking 4-23
Locking Modes 4-24
File/Record Locking Interaction 4-24
Deadlock 4-26
Record Locking with Unstructured Files 4-26
Record Locking Limitation....4-27
Purge Data 4-27
Verify Write 4-28
Refresh 4-28
Programmatic Extent Allocation 4-29
Extent Allocation Errors 4-29
Programmatic Extent Deallocation 4-31
Summary of Disc Control and Setmode Operations 4-32
Errors and Error Recovery 4-34
File Management Errors 4-34
Path Errors 4-35
Data Errors 4-35
Device Operation Error 4-36
Failure of Primary Application Process 4-36
Errors Grouped by Error Number 4-36
Special Considerations for Errors 200, 210, and 211 4-38
Error Recovery 4-39
Error Handling for Structured Files 4-40
For Key-Sequenced Files 4-42
For Files with Alternate Keys 4-42
For Partitioned Files 4-42
Action of Current Key, Key Specifier, and Key Length 4-43
Access Examples 4-47

SECTION 5. ENSCRIBE FILE CREATION 5-1
Considerations for Both Structured and Unstructured Files .5-2
File Type 5-2
Key-Sequenced Files 5-2
Relative Files 5-2
Entry-Sequenced Files 5-2
Unstructured Files 5-3
Extents 5-3
File Code 5-3
Partitioned (Multi-Volume) Files 5-4
Considerations for Structured Files 5-5
Logical Records 5-5
Blocks 5-5
Considerations for Key-Sequenced Files 5-6
Compression.....5-6
Primary Key 5-7
Index Blocks 5-8
Considerations for Files Having Alternate Keys 5-8
Key Specifier 5-8
Key Offset and Length 5-9
Null Value 5-9
No Automatic Update 5-10
Unique Alternate Keys 5-10
Alternate Key Files 5-11
The File Utility Program (FUP) 5-12
Running FUP 5-13
FUP SET Command 5-14
FUP SHOW Command 5-23
FUP CREATE Command 5-24
FUP RESET Command 5-25
FUP INFO Command 5-26
CREATE Procedure 5-28
Creation Examples 5-40
Example 1. Key-Sequenced File 5-40
Example 2. Key-Sequenced File With Alternate Keys 5-42
Example 3. Alternate Key File 5-44
Example 4. Relative, Partitioned File 5-45
Example 5. Key-Sequenced, Partitioned File 5-46

Table of Contents

SECTION 6. ENSCRIBE FILE LOADING 6-1
The FUP LOAD Command 6-2
The FUP LOADALTFILE Command 6-6
The FUP BUILDKEYRECORDS Command 6-9
File Loading Examples 6-12
Example 1. Load a Key-Sequenced File 6-12
Example 2. Add an Alternate Key to a File Having an Alternate Key 6-13
Example 3. Add an Alternate Key to a File Not Having Alternate Keys 6-14
Example 4. Reload a Single Partition of Key-Sequenced, Partitioned File 6-15
Example 5. Load a Single Partition of Partitioned Alternate Key File 6-16

APPENDIX A. SEQUENTIAL I/O PROCEDURES A-1
Source Files A-4
CHECK^BREAK Procedure A-5
CHECK^FILE Procedure A-6
CLOSE^FILE Procedure A-14
GIVE^BREAK Procedure A-16
OPEN^FILE Procedure A-17
READ^FILE Procedure A-23
SET^FILE Procedure A-25
TAKE^BREAK Procedure A-35
WAIT^FILE Procedure A-36
WRITE^FILE Procedure A-38
Sequential I/O Procedure Errors A-40
FCB Structure A-43
Initializing the File FCB A-44
Interface With INITIALIZER and ASSIGN Messages A-48
INITIALIZER-Related Defines A-48
Usage Example With the INITIALIZER A-52
Usage Example Without INITIALIZER A-56
NO^ERROR Procedure A-58
$RECEIVE Handling A-62
NOWAIT I/O A-65
$SYSTEM.SYSTEM.GPLDEFS A-66
File Control Block Format A-70

APPENDIX B. ASCII CHARACTER SET B-1

APPENDIX C. STRUCTURED FILE BLOCK STRUCTURE C-1

APPENDIX D. FILE MANAGEMENT ERROR LIST D-1

APPENDIX E. ENSCRIBE PROCEDURE SYNTAX SUMMARY E-1

LIST OF FIGURES

Figure 1-1. Disc File Organization 1-2
Figure 1-2. Key-Sequenced File Structure 1-4
Figure 1-3. Relative File Structure 1-5
Figure 1-4. Entry-Sequenced File Structure 1-6
Figure 1-5. Access Paths 1-9
Figure 1-6. Approximate, Generic, and Exact Subsets 1-10
Figure 1-7. Relational Access Among Structured Files 1-11
Figure 1-8. Wait Versus No-Wait I/O 1-18
Figure 1-9. No-Wait I/O (Multiple Concurrent Operations) 1-19
Figure 1-10. Hardware I/O Structure 1-26
Figure 1-11. Primary and Alternate Communication Paths 1-28
Figure 1-12. File System Procedure Execution 1-29
Figure 1-13. File Open 1-32
Figure 1-14. File Transfer 1-34
Figure 1-15. Buffering 1-35
Figure 1-16. Mirror Volume 1-44

Figure 2-1. Key-Sequenced File Structure 2-4
Figure 2-2. Relative File Structure 2-6
Figure 2-3. Entry-Sequenced File Stucture 2-7
Figure 2-4. Key Fields and Key Specifiers 2-9
Figure 2-5. Current Position 2-10
Figure 2-6. Alternate Key Position 2-13

Figure 3-1. AWAITIO Operation 3-20
Figure 3-2. File Security Checking 3-94

Figure 4-1. Example of Encountering EOF 4-14
Figure 4-2. Example of Encountering EOF (short read) 4-15
Figure 4-3. Example of File Pointer Action 4-17
Figure 4-4. Example of Crossing Sector Boundries 4-19
Figure 4-5. Resident Buffering 4-20
Figure 4-6. Example Showing Extent Allocation Error 4-30

Figure A-1. FCB Linking A-43

LIST OF TABLES

Table 2-1. File Pointer Action 2-19

Table 3-1. File Management Call Summary 3-1
Table 3-2. AWAITIO Action 3-19
Table 3-3. CONTROL Operations 3-25
Table 3-4. <key-sequenced params> Array Format 3-32
Table 3-5. <alternate key params> Array Format.....3-33
Table 3-6. <partition params> Array Format 3-36
Table 3-7. Device Types and Subtypes 3-40
Table 3-8. Exclusion/Access Mode Checking 3-95
Table 3-9. SETMODE and SETMODENOWAIT Functions 3-127

Table 4-1. File Pointer Action 4-11
Table 4-2. Disc CONTROL and SETMODE Operations 4-32
Table 4-3. Action of Current Key, Key Specifier, and Key Length 4-43

SYNTAX CONVENTIONS IN THIS MANUAL

The following is a summary of the characters and symbols used in the syntax notation in this manual. For distinctiveness, all syntactical elements appear in a typeface different from that of ordinary text.

Notation	Meaning
UPPER-CASE CHARACTERS	Upper-case characters represent keywords and reserved words. If a keyword is optional, it is enclosed in brackets. If a keyword can be abbreviated, the part that can be omitted is enclosed in brackets.
<lower-case characters>	Lower-case characters enclosed in less than/greater than symbols represent all variable entries supplied by the user. If an entry is optional, it is enclosed in brackets.
Brackets []	Brackets enclose all optional syntactic elements. A vertically-aligned group of items enclosed in brackets represents a list of selections from which one, or none, may be chosen.
Braces {}	A vertically-aligned group of items enclosed in braces represents a list of required elements from which exactly one must be chosen.
Ellipses ...	An ellipsis (...) following a pair of brackets that contains a syntactic element preceded by a separator character indicates that that element may be repeated a number of times. An ellipsis following a pair of braces that contains a series of syntactic elements preceded by a separator character indicates that the entire series may be repeated, intact, a number of times.
Punctuation	All punctuation and symbols other than those described above must be entered precisely as shown. If any of the above punctuation appears enclosed in quotation marks, that character is not a syntax descriptor but a required character, and must actually be entered.

Conventions for Procedure Calls

Calls to operating system procedures are shown in the following form:

```
{CALL     }  <procedure name> (<parameters>)
{<retval>:=}
```

where

CALL

is an ENSCRIBE CALL statement.

<retval> :=

indicates that the procedure is a function procedure; it returns a value of type INT or INT(32) when you reference it in a statement. All function procedures can be called with a CALL statement, but the return value will be lost. The return value is described as:

<retval>,<type>

where

<type>

is INT or INT(32)

<procedure name>

is the name of the procedure.

<parameters>

are described as:

```
<parameter>,<type> :{ ref   } [ :<num elements> ],
                    { value }
```

where

<type>

is INT,INT(32), or STRING.

ref

indicates a reference parameter.

value

indicates a value parameter.

<num elements>

indicates that the procedure returns a value of type to parameter for num elements.

SECTION 1

INTRODUCTION TO ENSCRIBE

ENSCRIBE is a data base record manager that provides high level access to, and manipulation of, records in a data base.

ENSCRIBE operates as an integral part of the GUARDIAN Operating System in a distributed fashion across multiple processors. As such, ENSCRIBE ensures the integrity of the application's data in the event that a processor module, i/o channel, or disc drive fails.

Important Features of ENSCRIBE are listed below:

- Four disc file structures
  - Key-Sequenced
  - Relative
  - Entry-Sequenced
  - Unstructured
- Multi-key Access to Records
- Relational Access Among Files
- Record Locking
- Sequential Access Buffering Option
- Record Management Procedures
- Automatic Maintenance of All Keys
- Data Compression for Key-Sequenced Files
- Index Compression
- Multiple-Volume (partitioned) Files
- Cache Buffer

DISC FILE ORGANIZATION

A disc file is referenced by the symbolic file name that is assigned when the a file is created. The symbolic name that identifies an individual disc file in the system consists of three parts:

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

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

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

This disc file organization is illustrated in Figure 1-1 below.

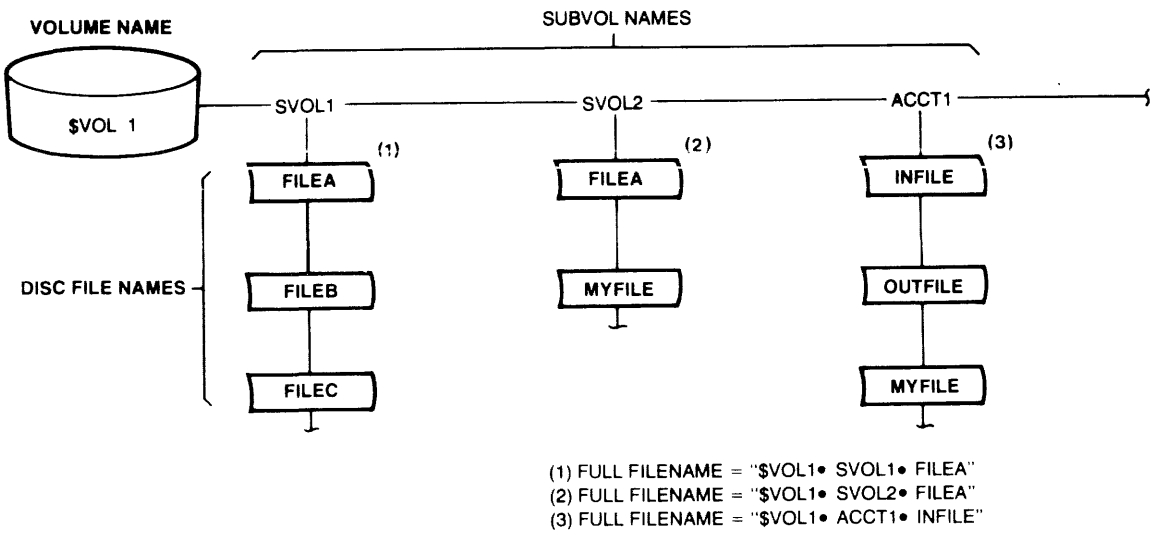


Figure 1-1. Disc File Organization

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

Also specified when a file is created is the file's type. ENSCRIBE supports four file types: key-sequenced, relative, entry-sequenced, and unstructured files. Taken as a group, key-sequenced, relative, and entry-sequenced files are known as structured files. The facilities available with structured files differ significantly from those available with unstructured files. The following sections briefly describe each of the four file types, beginning with the structured files.

STRUCTURED FILES

All data transfers between an application process and a structured disc file are done in terms of logical records. The placement of and access to records in a disc file is determined by the file structure (a file's structure is specified at file creation time).

For structured files, the maximum length of a logical record (i.e., the maximum number of bytes that can be inserted in a single operation) is specified for each file at file creation time. The actual number of bytes comprising a logical record can be variable (up to the specified record length); the minimum number of bytes that can be inserted depends on the file structure.

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

Key-Sequenced File Structure

Records are stored in ascending sequence according to the value of a field within each record called the "primary key field". The primary key field is designated when a key-sequenced file is created and may be any set of contiguous bytes within the data record. Physical and logical record lengths can be variable; a record occupies only the amount of space specified for it when inserted into the file.

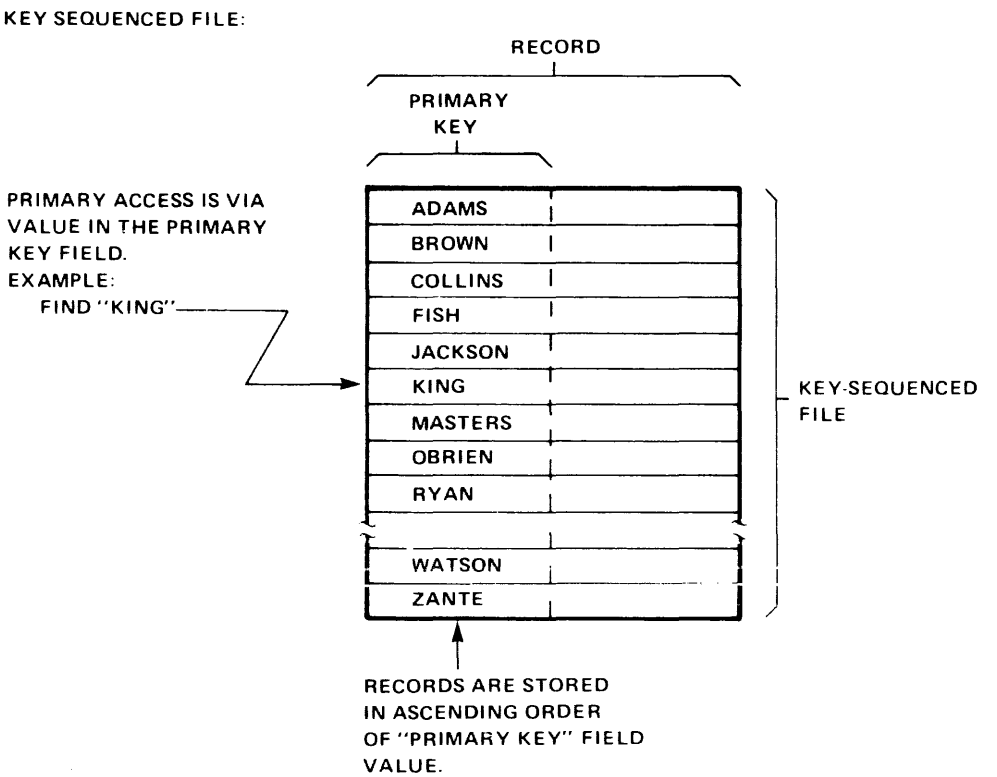


Figure 1-2. Key-Sequenced File Structure

Relative File Structure

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

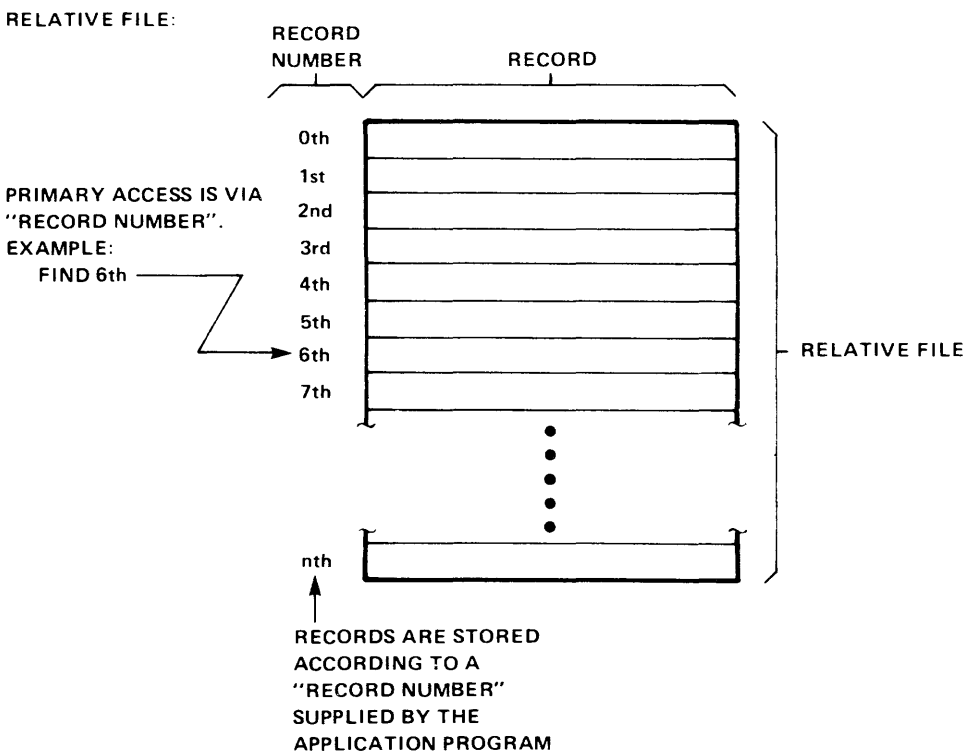


Figure 1-3. Relative File Structure

Entry-Sequenced File Structure

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

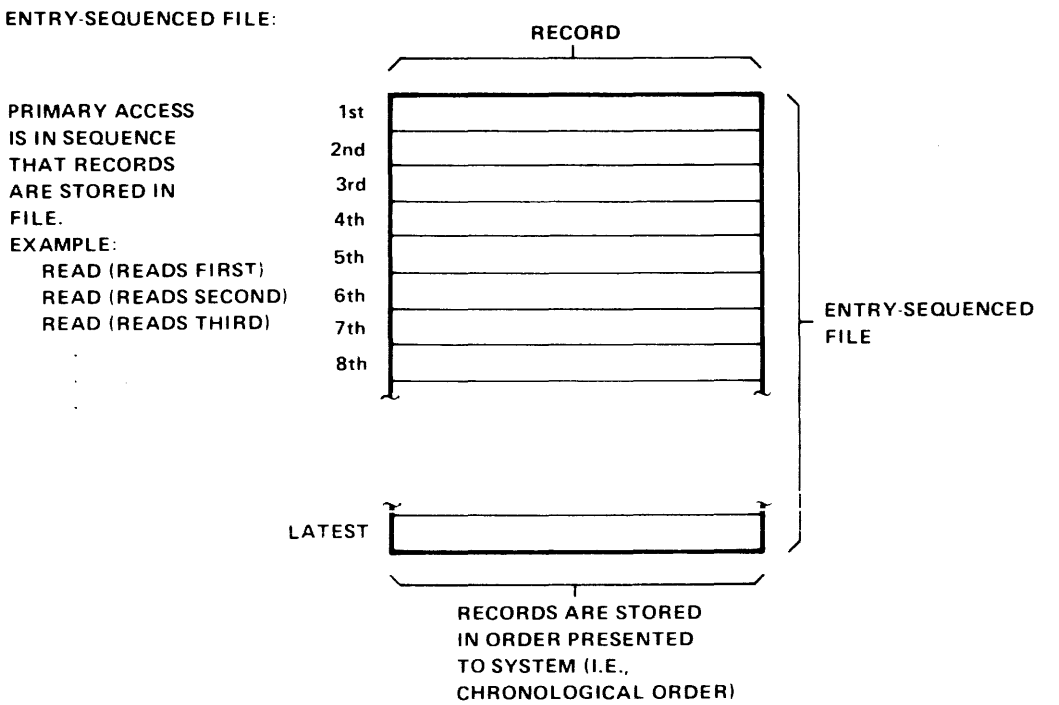


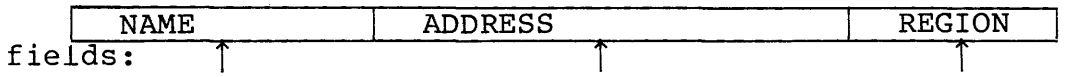
Figure 1-4. Entry-Sequenced File Structure

Multi-key Access to Structured Files

A "record" consists of one or more "fields":

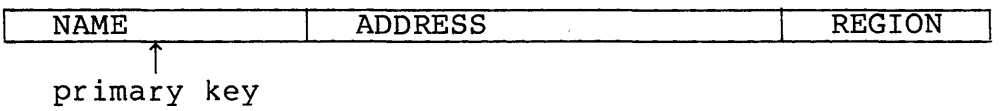
A record in a key-sequenced file:

CUSTOMER

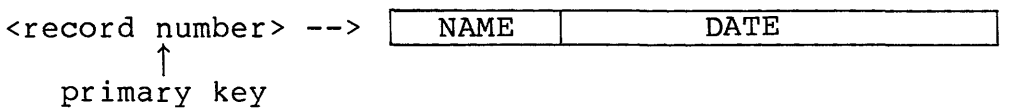


Each record in a file is uniquely identified among other records in that file by the value of its primary key. For key-sequenced files, the primary key is a byte field within a record; for relative files, the primary key is a "record number"; for entry-sequenced files, the primary key is a "record address".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

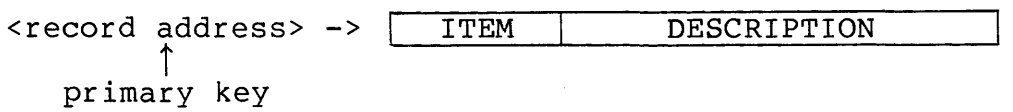
The primary key field for a key-sequenced file:



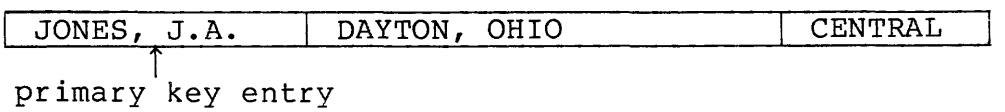
The primary key for a relative file:



The primary key field for an entry-sequenced file:



A record is located among records of the same file by the value of its primary key:



This is the only record of this record type having the primary key entry "JONES, J.A."

One or more byte fields within a record may be designated "alternate keys". Any structured file can have up to 255 alternate key fields. Values in alternate key fields need not be unique.

NAME	ADDRESS	REGION

an alternate key

Several associated records of the same type may be located by their entries in an alternate key field:

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

Two records of this record type have alternate key field entries of "CENTRAL".

Each key in a structured file provides a separate access path through records in that file. Records in an access pass are logically ordered by ascending access path key values.

A simple employee file with three seperate access paths, provided by three different key fields, is shown in Figure 1-5 on the following page.

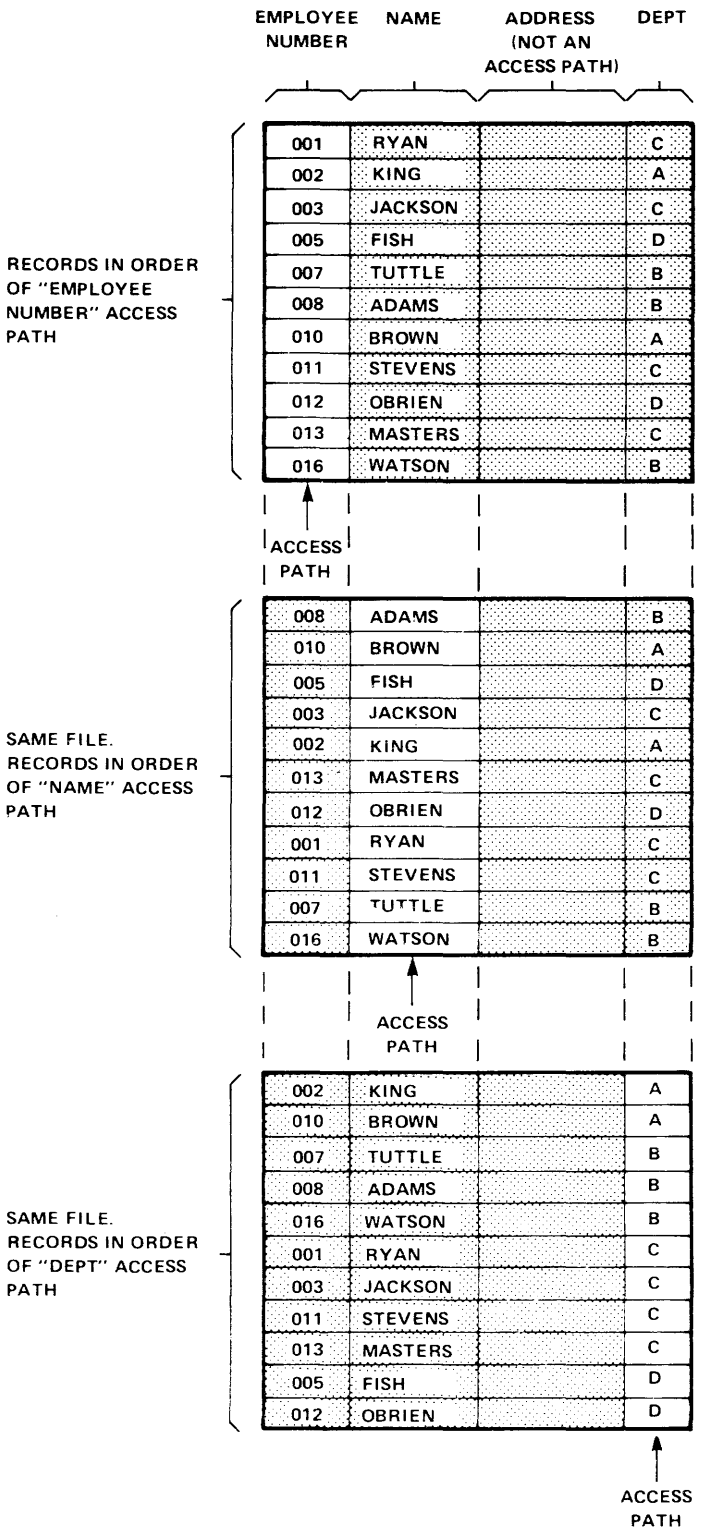


Figure 1-5. Access Paths

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 means that the subset comprises all records whose access path key value is equal to or greater than the supplied key value. Generic means that the subset is comprised of all records whose access path key value matches a supplied partial value. Exact means that the subset is comprised of only those records whose access path key value matches the supplied key value exactly. Examples of subsets returned with these three positioning modes are shown in Figure 1-6 below.

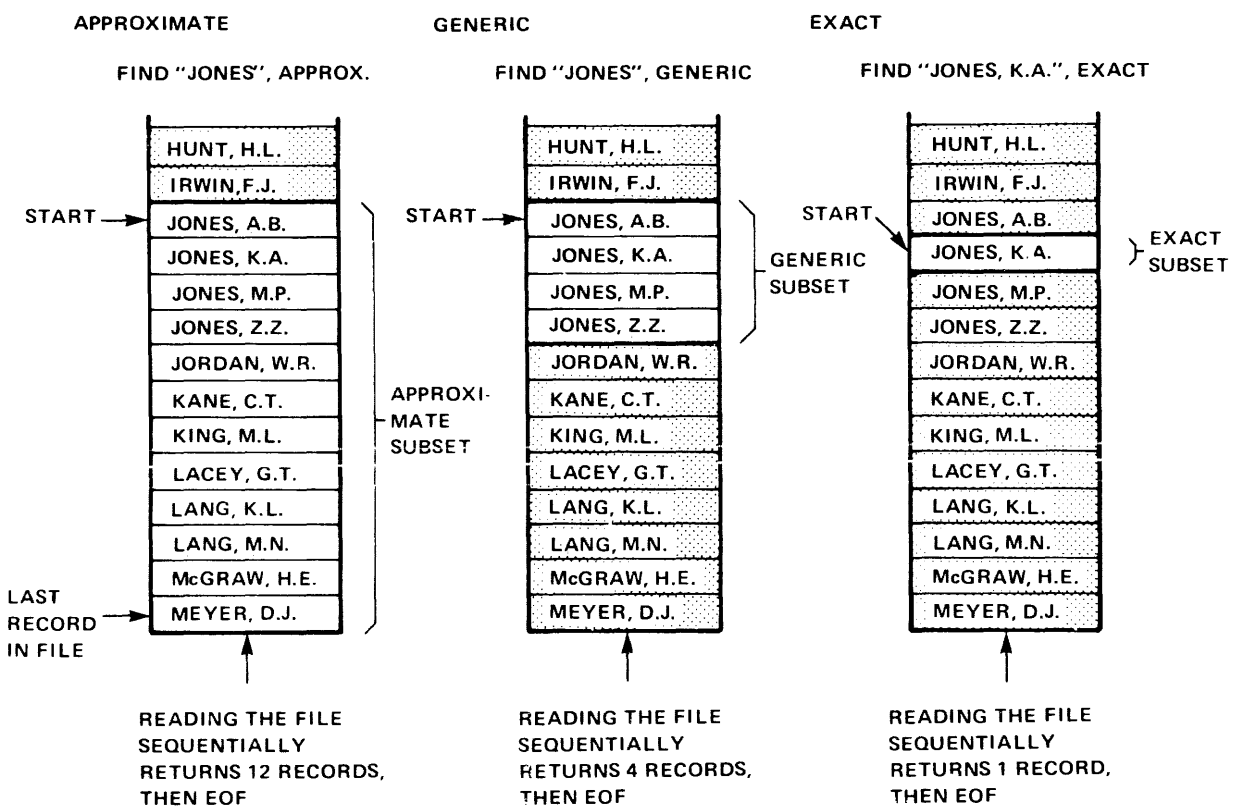


Figure 1-6. Approximate, Generic, and Exact Subsets

Relational Access Among Structured Files

Relational access among structured files in a data base is accomplished by obtaining a value from a field in a record in one file and using that value to locate a record in another file. An example of relational access is shown in Figure 1-7 below.

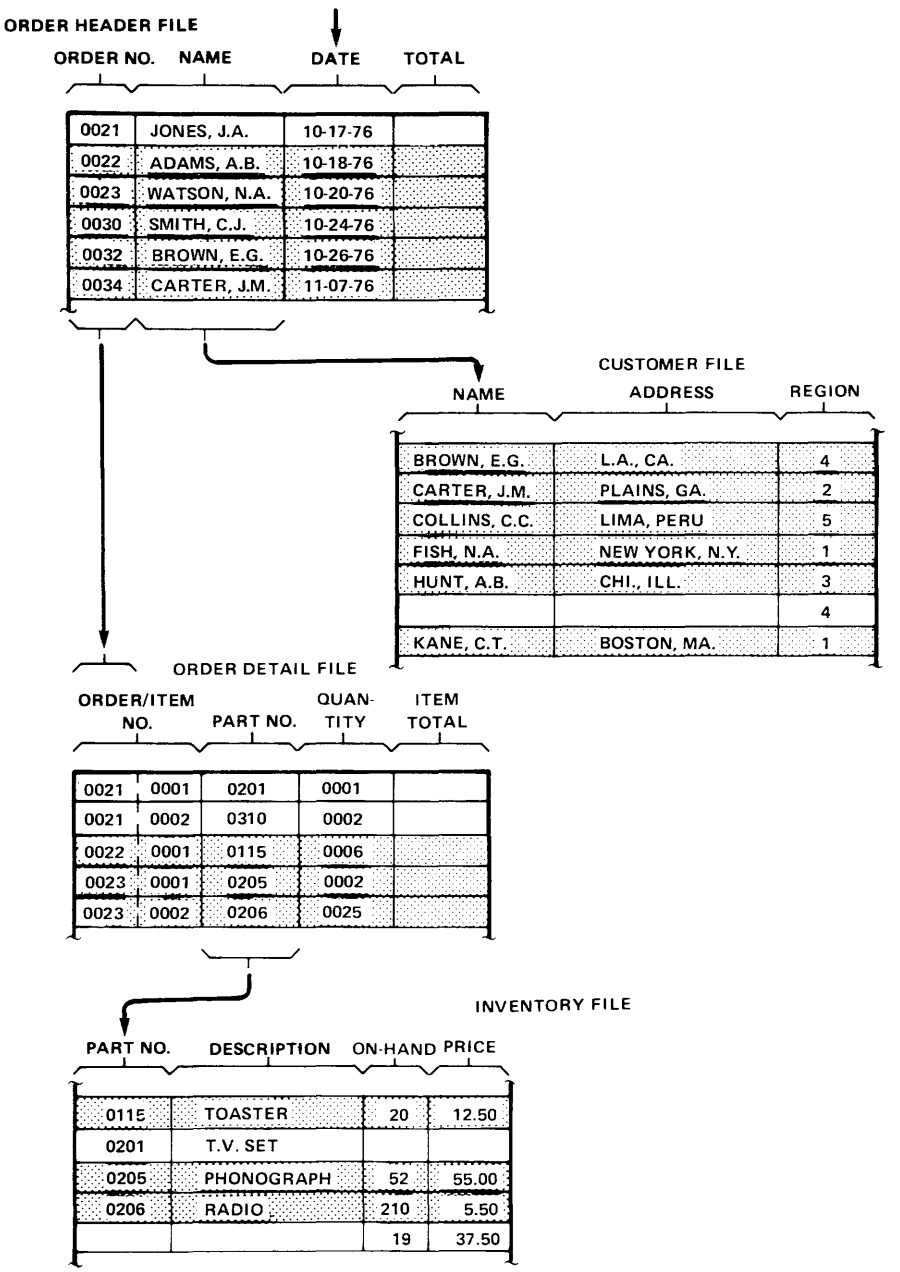


Figure 1-7. Relational Access Among Structured Files

Automatic Maintenance of All Keys

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

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

Data and Index Compression

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

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

Both data and index compression may be specified for a file when the file is created.

UNSTRUCTURED DISC FILES

An unstructured disc file is essentially a byte array. The organization of an unstructured disc file, the lengths and locations of records within the file, is the responsibility of the application process.

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

Associated with each open unstructured disc file are three "pointers":

- a "current-record" pointer
- a "next-record" pointer
- an "end-of-file" pointer

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

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

ACCESS COORDINATION AMONG MULTIPLE ACCESSORS

A file may be accessed by several different processes at the same time. In order to coordinate simultaneous access, each process must indicate when opening the file how it intends to use the file. Both an "access mode" and an "exclusion mode" must be specified.

The access mode specifies the operations that will be performed by an accessor. The access mode is specified as one of

- Read/Write (default access mode)
- Read-Only
- Write-Only

The exclusion mode is used by a process to specify the type of access it can tolerate by other accessors. The exclusion mode is specified as one of

- Share Access (default exclusion mode)

  Share access indicates that the opening process can tolerate simultaneous read and/or write access to the file.

- Exclusive Access

  Exclusive access indicates that the opening process cannot tolerate any simultaneous access of any kind to the file. Therefore, if any further opens are attempted while the file is open, they are rejected. Likewise, if the file is already open, an open specifying exclusive access is rejected.

- Protected Access

  Protected access indicates that the opening process can tolerate a simultaneous read access to the file but cannot tolerate a simultaneous write access to the file. Therefore, if any further opens that specify read/write or write-only access mode are attempted while the file is open, they are rejected. Likewise, if the file is already open with read/write or write-only access mode, an open specifying protected access is rejected. However, a simultaneous open that specifies read-only access mode is permitted.

LOCKING

The access and exclusion mode operate on a file from the time it is opened until the time it is closed. To prevent concurrent access to a disc file for shorter periods of time, a locking mechanism is provided. Two types of locking are available, file locking and record locking.

File Locking

File locking is accomplished with the LOCKFILE and UNLOCKFILE procedures. Multiple processes accessing the same disc file call LOCKFILE before performing a critical sequence of operations to that file. If the file is not currently locked, it becomes locked and the process continues executing. This prevents other accesses to the file until it is unlocked through a call to UNLOCKFILE. If the file is locked, a caller of LOCKFILE is suspended until the file is unlocked. If a process attempts to write to a locked file, the access is rejected with a "file is locked" error indication; if a process attempts to read from a locked file, it is suspended until the file is unlocked.

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

Record Locking

Record locking for ENSCRIBE disc files is accomplished through the LOCKREC and UNLOCKREC procedures. Record locking operates in essentially the same manner as file locking, however it allows a greater degree of concurrent access to a single file than file locking. Through a call to LOCKREC, a process locks the current record (as indicated by the last operation with the file). If the record is currently locked, then the caller of LOCKREC is suspended until the record is unlocked. Records are unlocked by a call to UNLOCKREC.

Both record and file locking may be done concurrently to the same file. A file lock will wait for all records to be unlocked before it will be granted. Similarly, a record lock must wait if the file is currently locked.

WAIT/NO-WAIT I/O

The file system provides the capability for an application process to execute concurrently with its file operations.

Two definitions:

- Wait I/O (the default)

  "Wait" i/o means that when designated file operations are performed (i.e., via file management calls), the application process is suspended, waiting for the operation to complete.

- No-wait I/O

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

Whether "wait" or "no-wait" i/o is to be in effect when designated file operations are performed is specified on a per OPEN basis when files are opened. If "no-wait i/o" is specified, then the maximum number of concurrent operations to be permitted must also be specified at file open. Disc files are limited to one concurrent operation (one outstanding no-wait call) per file open.

For example, to open a file so that one concurrent file operation is permitted (i.e., a "no-wait" file), the following could be written in an application program (assume that "filename" contains a valid file name):

```
CALL OPEN (filename, fnum^1, 1);
```

The third parameter, "1", specifies that one concurrent operation is permitted (this parameter is also used for other purposes, see "OPEN").

Any input/output operation involves an "initiation" and a "completion". With "wait" files both the initiation and completion are performed in the same file management procedure call. For example, on a wait file, the call

```
.
CALL READ(fnum^0,buffer,..);
```

initiates the i/o operation, then the application process is suspended, waiting for its completion.

With "no-wait" files, the "initiation" is performed in one call, the operation is "completed" by another call:

```
.
CALL READ(fnum^1,buffer,..);
.
```

initiates the i/o operation. Process execution continues concurrently with the i/o transfer.

```
.
CALL AWAITIO(fnum^1,..);
.
```

completes the i/o operation. If not complete when AWAITIO is called, the process is suspended until completion occurs or an application-defined timeout expires.

Multiple operations (with multiple files) can be in progress simultaneously. Concurrent operations associated with a particular open are "completed" in the same order as initiated, unless setmode 30, "Allow no-wait operations to complete in any order", has been specified (GUARDIAN operating system version D or later). Concurrent operations associated with separate opens are "completed" as they finish.

The difference between wait and no-wait i/o is illustrated in Figure 1-8, on the following page.

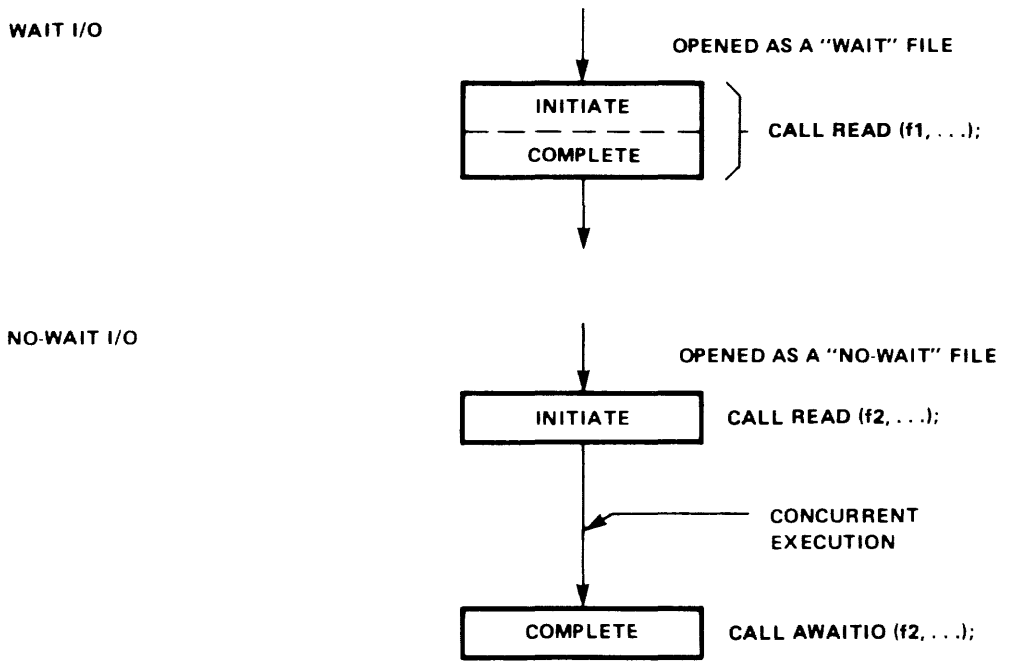


Figure 1-8. Wait Versus No-Wait I/O

The action of no-wait i/o during multiple concurrent operations is shown in Figure 1-9, on the following page.

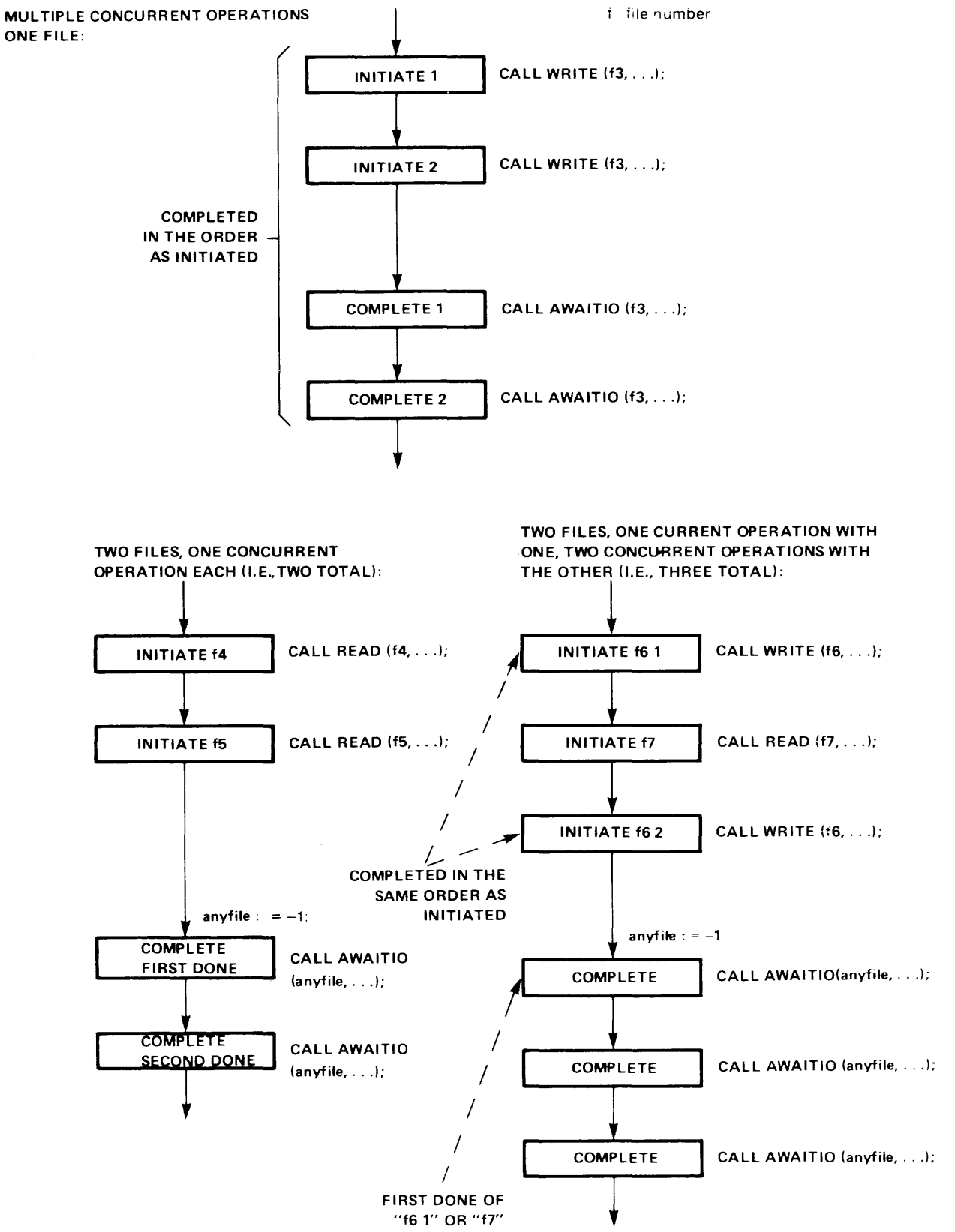


Figure 1-9. No-Wait I/O (Multiple Concurrent Operations) 0878

CACHE

The "Cache" is an area of main memory, whose size is specified at system generation time, that is reserved for buffering blocks read from disc.

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

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

When a request is made to write a record to disc, the block in the Cache that contains the record is modified then immediately written to disc (if the block to be modified is not in the Cache, it is first read from the disc). However, the modified block remains in the Cache until it is needed for overlay.

SEQUENTIAL ACCESS BUFFERING OPTION (Structured Files Only)

For a program that sequentially reads a file, the access time to individual records can be greatly reduced by means of the Sequential Access Buffering Option (this option, if desired, is specified at file open). Basically, this option allows the record deblocking buffer to be located in the application process's data area (rather than in a system i/o process). This buffer is then used by ENSCRIBE to deblock the file's records. The advantage to this buffering is that it eliminates the request to the system i/o process to retrieve each record in a block (instead, a request retrieves an entire block of records). This option is allowed only if the file is opened by the requesting process with protected or exclusive access.

MULTIPLE-VOLUME (PARTITIONED) FILES

At file creation time, a file can be designated to reside entirely on a single volume or may be partitioned to reside on separate volumes. Moreover, the separate volumes need not reside on the same system; a file can be partitioned accross network nodes. Up to sixteen partitions are permitted; each partition can have up to sixteen extents.

In addition to providing a maximum file size of approximately four billion bytes, the use of multi-volume files provides for simultaneous access to a file's records:

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

FILE CREATION

Disc files are created (defined) by

- using the Tandem-supplied File Utility Program (FUP)

  The "creation parameters" (i.e., file type, record length, key description, etc.) for a file to be created are set (specified) by entering FUP commands. The state of file creation parameters can be displayed and modified before the file is actually created. Creation parameters can be set to those like another, existing file.

  FUP accepts commands entered at an online terminal or from a file such as an EDIT-format file.

- calling the File Management CREATE procedure

  programmatic file creation of disc files is accomplished by supplying the appropriate parameters to the CREATE procedure.

File creation is described in section 5, "ENSCRIBE File Creation".

DATA DEFINITION LANGUAGE (DDL)

The Data Definition Language (DDL) provides a uniform method of describing record formats, regardless of the programming language used (COBOL, FORTRAN, or TAL) to access the record. DDL also provides a system-wide definition of record formats so all programs have a consistent definition of a given record format. (See the Data Definition Language Programming Manual, T16/8034.) In addition to data language source, DDL can produce FUP File Creation commands for data base files which are then accessible through ENSCRIBE. See File Creation described in Section 5.

FILE LOADING

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

File loading is described in section 6, "ENSCRIBE File Loading".

RECORD MANAGEMENT FUNCTIONS

Manipulation of records in an ENSCRIBE file is performed by calling File Management Procedures. Record management functions and their associated procedures are:

Function	Description	Procedure
• Find.	Set the current position, access path, and positioning mode for a file. This may indicate the starting record of a subset of records in anticipation of a sequential read of the set, or may specify a record for a subsequent update	KEYPOSITION, POSITION
• Insert	a new record into a file according to its primary key value	WRITE
• Read	a subset of records sequentially	READ
• Update	a record in a random position in a file	READUPDATE, WRITEUPDATE
• Delete	the record in a key-sequenced or relative file as indicated by a primary key value	WRITEUPDATE
• Lock	the current record in a file, or the file	LOCKREC, LOCKFILE, READLOCK, READUPDATE-LOCK
• Unlock	the current record in a file, or all records in the file	UNLOCKREC, UNLOCKFILE, WRITEUPDATE-UNLOCK
• Define	a new file	CREATE

The Record Management Procedures are described in section 3.

FILE SYSTEM IMPLEMENTATION

This description is intended to provide a basic understanding of the internal operation of the file system. In particular, the programmer should have a thorough understanding of the action that the file system takes when a communication "path" failure occurs and the corresponding action that the application program must take to recover.

Topics covered in this description are:

- File and I/O System Structure
- File System Procedure Execution
- File Open
- File Transfer
- File Close
- Automatic Path Error Recovery for Disc Files
- Mirror Volumes

File and I/O System Structure

The file and i/o structure encompasses the following areas:

- Hardware Structure

and

- Software Structure.

HARDWARE STRUCTURE. The hardware structure of a Tandem System is designed so that two physically independent communication "paths" exist between any application process and any i/o device.

The hardware communication path associated with an i/o operation is comprised of the following:

- The interprocessor buses

  Interprocessor buses are used for communicating data and control information between processor modules. (The interprocessor bus is not part of the communication path if the processor module controlling the device is same as one where the application process requesting an i/o operation is running)

- The processor module controlling the device

  The processor module controlling a device executes i/o instructions to command the device to perform designated i/o functions, contains the main memory where the i/o transfer takes places, and receives completion status from the hardware controller

- The i/o channel to which the device is connected

  The i/o channel carries the control and data signals between a processor module and i/o controllers. (Up to 32 controllers may be connected to a single channel)

- The i/o controller

  The i/o controller provides the electrical interface between and i/o device and the i/o channel. (I/O controllers are generally capable of controlling multiple devices)

Two physically independent communication paths are accomplished as follows:

- The two interprocessor buses provide two independent communication paths between processor modules. If either bus fails, the other is still available

- I/O controllers have two interface ports and are connected to the i/o channels of two processor modules. If one channel fails, control of the i/o controller is accomplished via the i/o channel connected to the other processor module.

The hardware i/o structure is shown in Figure 1-10 on the following page.

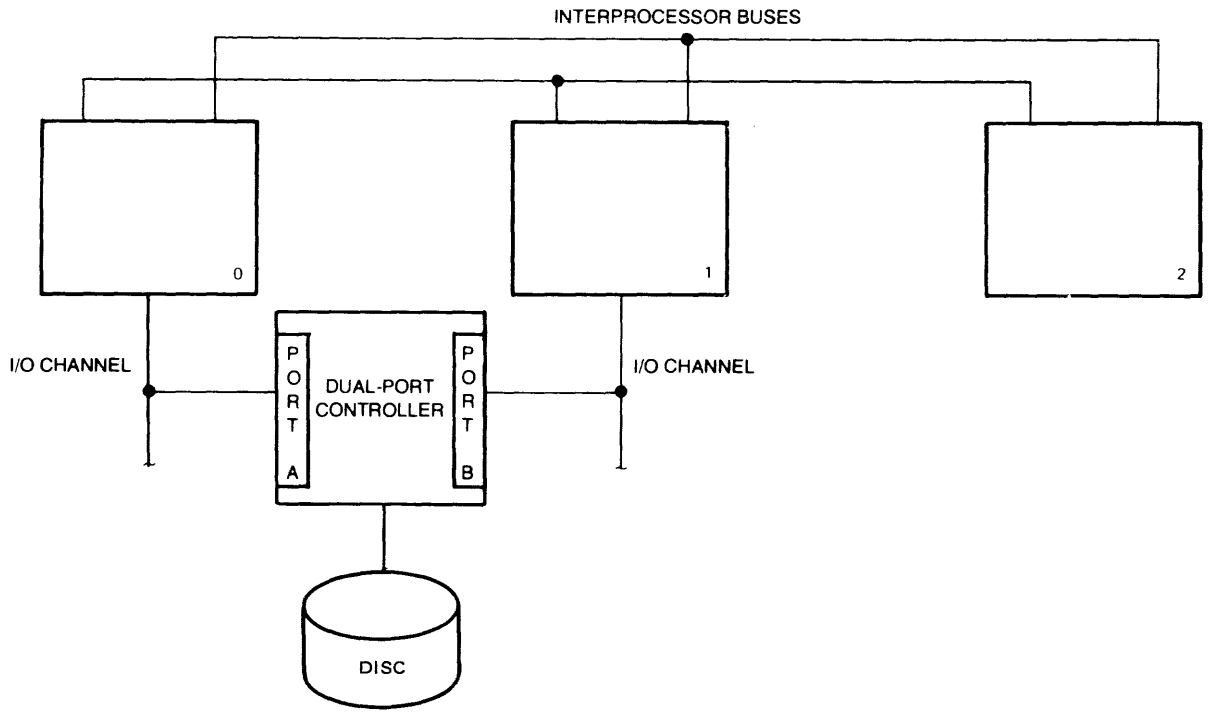


Figure 1-10. Hardware I/O Structure

SOFTWARE STRUCTURE. The GUARDIAN File Management System is designed so that if at any time during a file operation any part of a communication path fails, the file operation can still be completed successfully.

The file management system is an integral part of the GUARDIAN Operating System. A copy of the operating system resides in each processor module in the system. Each copy contains only what is necessary to control the input/output devices connected to its particular processor module.

System control of i/o devices is accomplished by means of "system i/o processes". The action of an i/o process is to accept a request from the file system (the request initially comes from an application process), perform the requested action (e.g., read or write), return the completion status of the operation (and also data if a read operation) to the file system, then wait for another request.

There are two system i/o processes for each device (or set of devices in the case of terminals or data communication lines); one located in each of the two processors which are physically connected to a given device. One process is designated the "primary" i/o process; the other is designated the "backup" i/o process. (This primary/ backup designation is made at system generation time.) Either i/o process is capable of controlling the device. However, they do not control the device simultaneously. Instead, the primary i/o process controls the device exclusively and, at the same time, keeps the backup i/o process informed (via checkpoint messages) of the activity on the device.

The communication path (i.e., processor module, i/o channel, and controller port) through a primary i/o process to the device that it controls is called the "primary path"; the path through through a backup i/o process is called the "alternate path". If the file system (or the operating system on behalf of the file system) detects a failure in the primary path, it shuts down the primary path and automatically reroutes subsequent communication to the device via the alternate path. The backup i/o process takes control of the device and, in fact, becomes the primary i/o process for the device.

In the case of disc files, the error recovery following a failure of the primary path is automatic, and this type of failure is completely invisible to the application program (see "Automatic Path Error Recovery for Disc Files" later in this description).

When the original primary path is restored to system operation, it becomes the current backup path. The original primary path is restored to primary operation for the following reasons: the system is cold loaded, a failure occurs in the current primary path, a PUP "PRIMARY" command is executed to switch control of the device, or "return to configured primary" is configured for the device and the original primary processor module is reloaded. (See the GUARDIAN Operating System Operating Manual for the NonStop system, or the GUARDIAN Operating System Management Manual for the NonStop II system, for an explanation of "cold load" and "reload."

Figure 1-11 below shows the primary and alternate communication paths to a device. While the primary path is operable, all i/o transfers occur via that path. Only when a failure of the primary path is detected, does the alternate path come into use. Once an alternate path is brought into use, it becomes the primary path and is used exclusively.

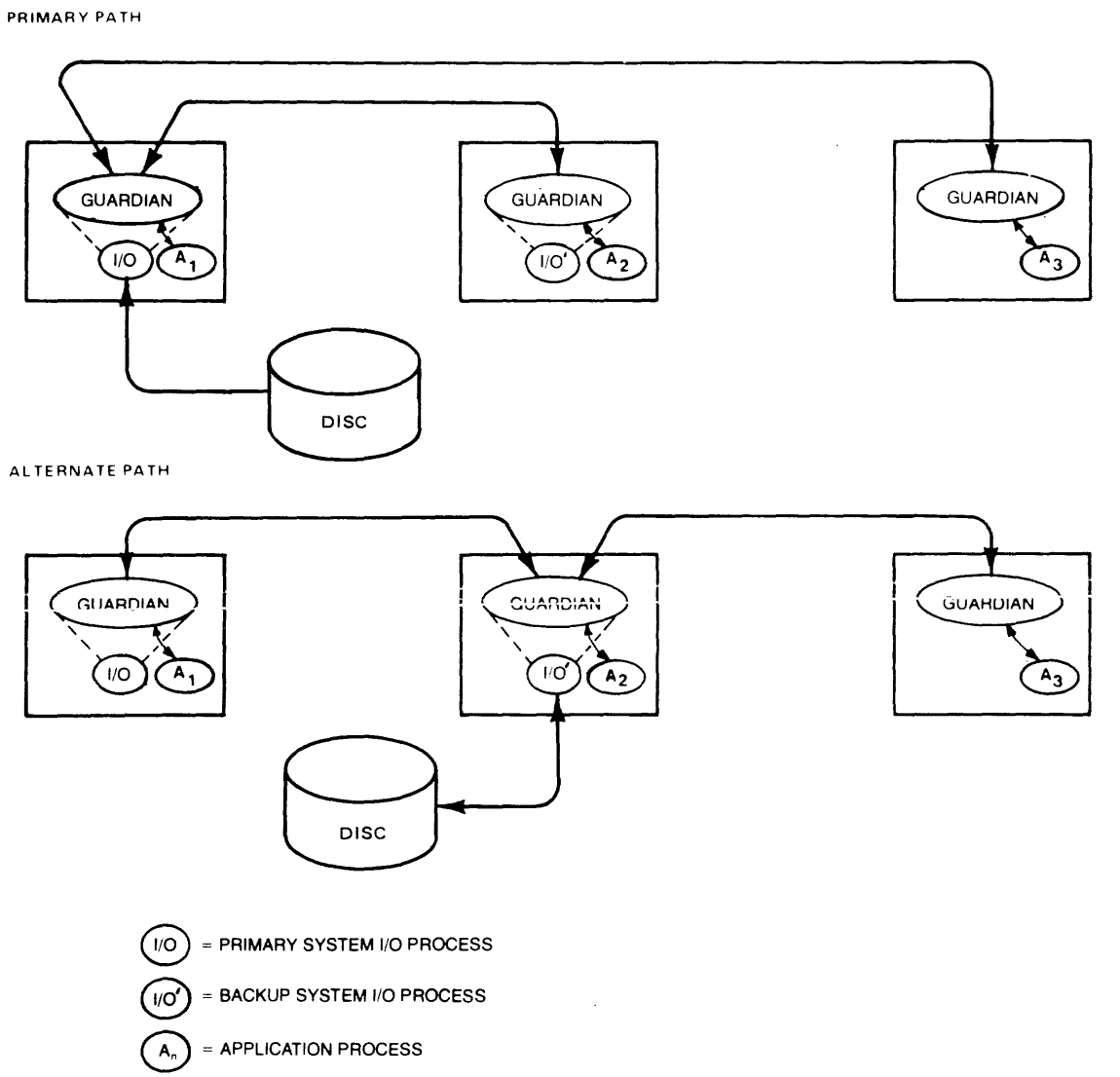


Figure 1-11. Primary and Alternate Communication Paths

File System Procedure Execution

File system procedures reside in operating system code but execute in the application process's environment. When a file system procedure (or any operating system procedure for that matter) is called by an application process, the system procedure's local storage is allocated in the application process's data stack. The maximum amount of local storage required by a call to a system procedure is approximately 400 words. (See Figure 1-12.)

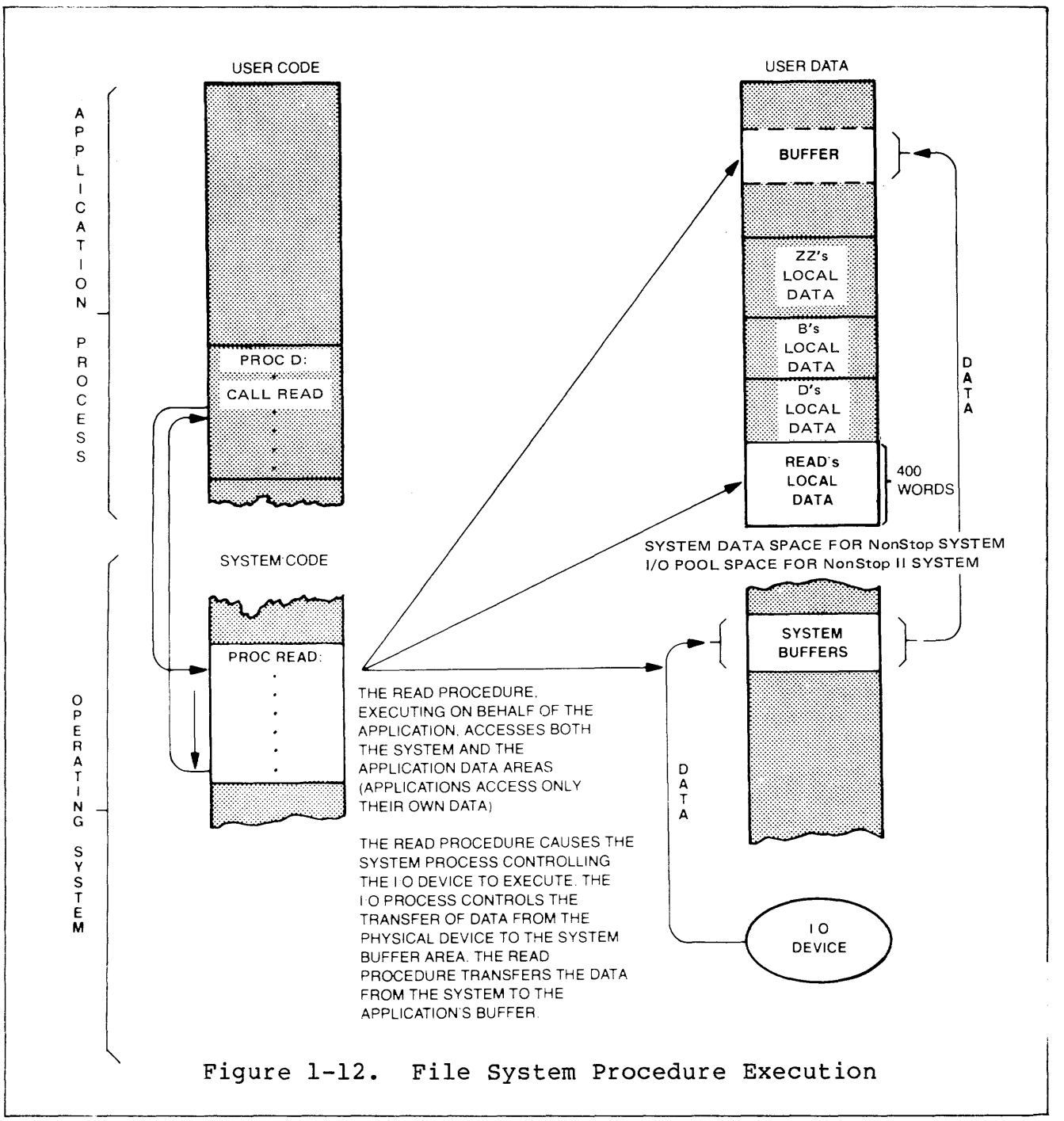


Figure 1-12. File System Procedure Execution

File Open

The OPEN procedure establishes a communication path to a file. The symbolic file name that identifies a file is used to search a table, a copy of which resides in each processor module, called the Logical Device Table. The Logical Device Table contains an entry for each device connected to the system. Each entry contains a device name or, in the case of disc files, a volume name, the process id of the "primary" system i/o process that controls the device/volume, and the process id of the "backup" system i/o process that controls the device/volume.

> In the following illustration, the logical device table is searched for an entry corresponding to the volume name "$VOLUME". The entry is associated with logical device four and a path is established to the primary i/o process controlling the device.

Next, in the case of disc files, a directory on the disc volume is searched for the subvolume name and the the disc file name that was supplied to OPEN. The entry associated with a subvolume and disc file name is a file label which contains information describing the state of the file, including the location of allocated extents, end-of-file location, file type, etc.

> In the following illustration, the file label is searched for a subvol designated "MYFILES" and a disc file named "FILEA".

Once the file is located, whether it is a disc file, non-disc device, process, or the operator console, an Access Control Block (ACB) is created for that file in the processor's memory where the caller to OPEN is running. The ACB is used by other file system functions when referencing the file. It contains information such as the logical device number of the device where the file resides and, for disc files, information "local" to the particular open of the file such as the current record pointer and next record pointer.

If the open is to a disc file and the file is not currently open, one File Control Block (FCB) is created in the memory of each of the two processor modules that contain the system i/o processes that control the volume containing the file. The FCB contains information which is "global" to all accessors of the file. This includes a copy of the information from the file label, such as allocated extents and end-of-file location, along with dynamic control information such as which process has the file locked and which processes if any are waiting to lock the file.

There is a single FCB for each open disc file in the system (in each of the two processor modules controlling the associated device) while there is an ACB created every time a file is opened. Thus each open of a given file provides a logically separate access to that file (i.e., separate current-record and next-record pointers), yet the end-of-file location is maintained in the FCB so that it has the same setting for all accessors of the file.

In the following illustration, the access control block indicates that logical device four is associated with the file indicated by "$VOL1".

When OPEN completes, it returns a file number to the application process. The file number is an index into a table that contains an address pointer to the associated access control block.

How the File System takes the file name passed with an OPEN procedure and builds the control blocks used to control subsequent access to the file is illustrated in Figure 1-13 on the following page.

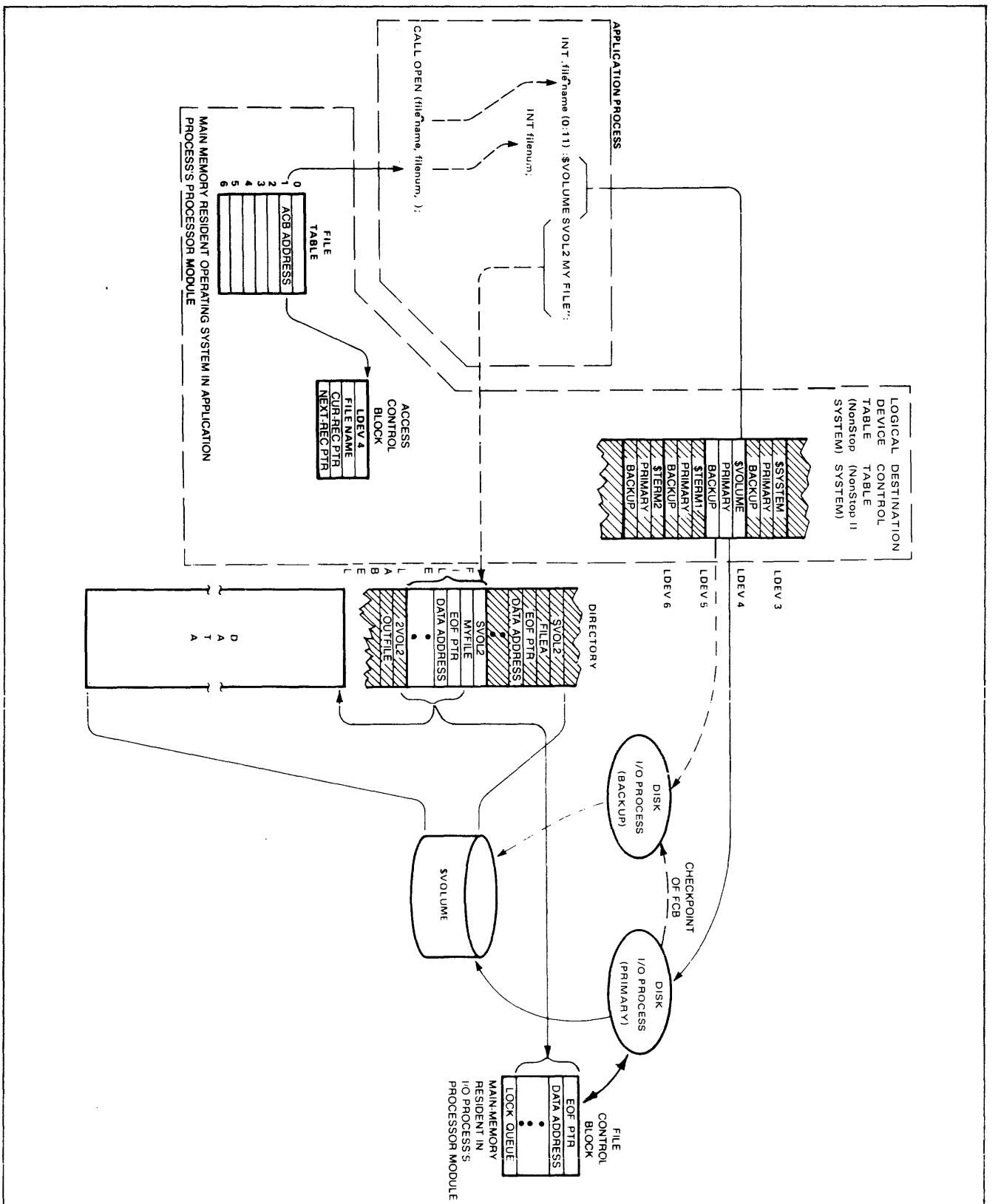


Figure 1-13. File Open

File Transfers

As previously mentioned, the file number returned from open is used by file management procedures to access an open file. The file number can be thought of as a pointer to an access control block. When performing an i/o operation, the file number is used to locate an access control block, which in turn provides a logical device number which is then used as an index into the logical device table. The corresponding entry in the logical device table provides the process id associated with the primary path to the physical i/o device. In the case of disc files, other information maintained in the access and file control blocks eliminates the need for a disc access when addressing a file.

> In the following illustration, Figure 1-14, the access control block indicates that the device is logical device four.

In the case of disc files, the information in the access control block (such as the current-record and next-record pointers) and the information in the file control block (such as the end-of-file pointer and addresses of allocated extents) is updated with the execution of each i/o operation.

As accesses which necessitate changes to the FCB are made to the file, the system process which is currently responsible for controlling the disc ensures that the copy of the FCB in the other processor which can access the disc is updated. Thus, if the primary processor fails, the backup has all the information necessary for a smooth transition (which is invisible to the user). In addition, when a new extent is allocated or the file is renamed, the file label on the disc is updated to reflect this change. This ensures that no disc space is lost, even in the event of a total system failure. However, when the end-of-file is changed or the file is written into, which requires updating the last modification timestamp, only the main-memory copies are updated. (Updating the file label each time the file is written into would be an unnecessary amount of additional overhead, because the current eof and last modification timestamp would be lost only in the event of a total system failure. The user who is concerned about the eof being updated on disc can force this to happen with the CONTROL request to set the end-of-file.)

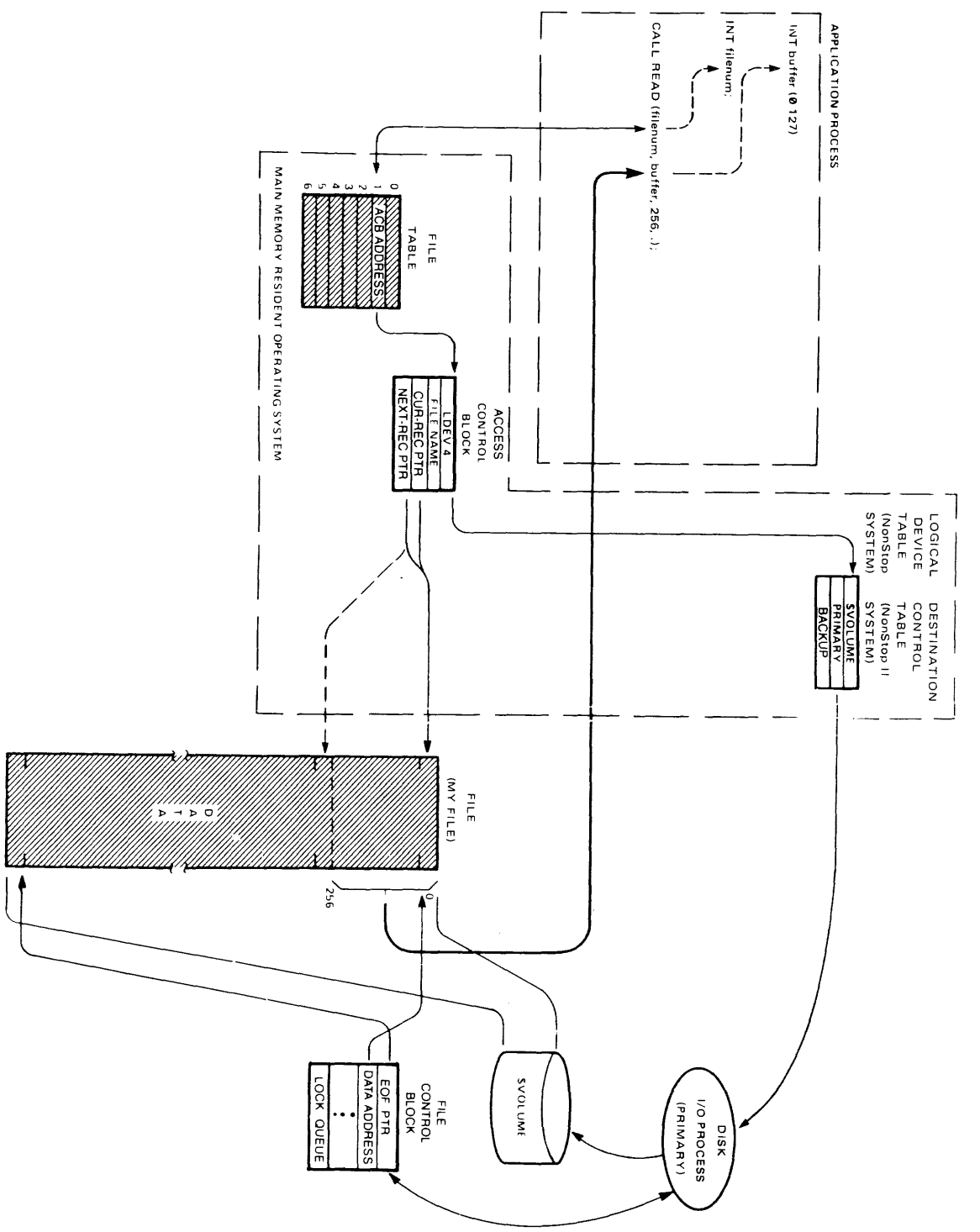


Figure 1-14. File Transfer

Buffering

Two operating system buffers and an application buffer are involved in an i/o transfer. The operating system buffers, designated File System Buffer and I/O Buffer, are shown in Figure 1-15 below.

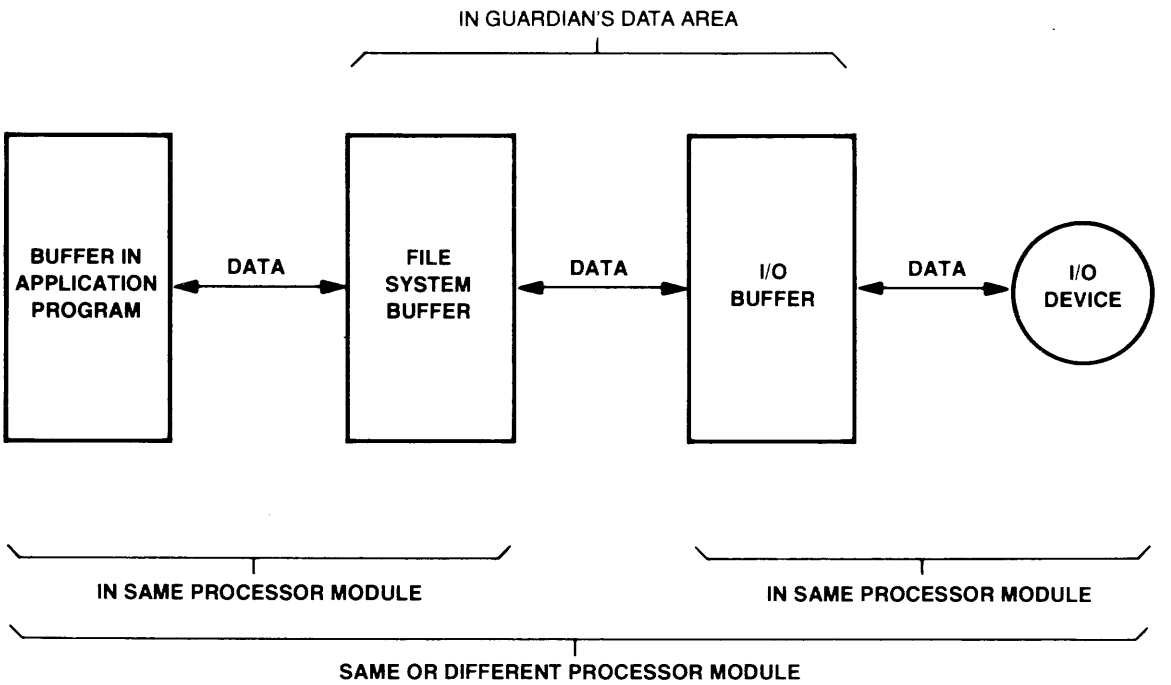


Figure 1-15. Buffering

When an i/o transfer is initiated, in this example a call to READ from a disc file, the file system first secures resident File System Buffer space in the processor module where the application process is executing. The amount of File System Buffer space secured is dependent on the transfer count specified in the file management procedure call. Next, the i/o process in the primary processor module controlling the disc is instructed (in this example) to read a block of data from the disc.

The i/o process first secures resident I/O Buffer space in its processor module (the amount of I/O Buffer space secured is dependent on the transfer count specified in the file management procedure call) then initiates the i/o transfer. When the i/o transfer with the device is completed, the data is moved from the I/O Buffer in the device's processor module to the File System Buffer in the application's processor module. If these are different processor modules, then this is accomplished by an interprocessor bus transfer. At this point, the file system (executing on behalf of the application process) moves the data from the resident File System buffer to an array in the application process's (virtual) data area.

For the Tandem NonStop II system, File System buffers are obtained from the process's Process File Segment (PFS). I/O buffers are obtained from the i/o segments as needed by the i/o process. Processes that require dedicated buffers obtain buffer space during initialization. Once a process has obtained dedicated buffer space, it keeps that space until it terminates execution.

For the Tandem NonStop System, File System Buffers are obtained from a memory space pool, called SHORTPOOL, in the operating system's data area. Processes requiring File System Buffers compete for this space on a first-come, first-served basis. If space is not available when needed, the application process is suspended until either the needed space becomes available or a configured timeout period expires (in the latter case an error indication is returned to the application process). When an i/o transfer is completed, the space in use by the File System Buffer is returned to SHORTPOOL for use in subsequent data transfers.

For the Tandem NonStop System, there are three types of I/O Buffers (the type of buffer that a device uses is specifed at system generation time):

- Pooled buffers - buffer space is secured from an i/o buffer pool, called IOPOOL, in the operating system's data area. I/O processes controlling devices using pooled buffers compete for space on a first-come, first-served basis. If space is not available when needed, the i/o process is suspended until either the needed space becomes available or a configured timeout expires (if a timeout occurs, an error indication is returned to the application process). When an i/o transfer is completed, the I/O Buffer space is returned to IOPOOL for use in subsequent data transfers.

- Shared buffers - buffer space in the operating system data area is shared among two or more i/o devices on the same controller.

- Dedicated buffers - buffers space in the operating system data area is dedicated to a device.

File Close

When a file is closed, the communication path to the file is broken. The access control block is deleted and the space that it used is returned for use as another access control block. In the case of disc files, if no other opens are outstanding for the file, then the file control block is also released and information such as the end-of-file pointer and addresses of allocated extents is updated on the physical disc from the information that was maintained in the file control block.

Automatic Path Error Recovery for Disc Files

The system accomplishes automatic path error recovery with disc files in the following manner.

First, two definitions: operations with disc files are classified as either "retryable" and "nonretryable". Retryable operations are those that can be retried indefinitely, without the possibility of loss or duplication of data. The retryable operations are: reading and full-sector writes. Nonretryable operations are those that if retried, could cause a loss or duplication of data. The nonretryable operations are: partial-sector writes and appending to the end-of-file.

Associated with each distinct file operation is a "sync id" and a "requestor id"; these are kept in the file's Access Control Block. The sync id identifies a single operation in a series of operations; the "requestor id" identifies the process requesting an i/o operation; together they identify a particular operation requested by a particular process. Additionally, each disc i/o process maintains a list of completed operations, each operation being identified by a sync id and a requestor id; these are kept in the File Control Block.

When an application program calls a file management procedure to write to disc, the file system initiates an i/o operation by sending an "i/o request" message to the primary i/o process for that file. The i/o request message contains the data be written, along with a sync id, the requestor id, and the address where the data is to be written.

The primary i/o process, upon receipt of the request, stores the information contained in the message and begins processing the request.

If the request involves a non-retryable operation (i.e., a partial-sector write and/or append to the end-of-file), special action is taken. The primary i/o process first reads the sector to be changed and updates the sector image in memory (if a partial-sector write). The primary i/o process then sends the new or updated sector image in a checkpoint message to its backup i/o process along with the disc address of where it is to be written, the sync id, and requestor id. Next, the primary i/o process performs the physical i/o operation to the disc. Upon completion of the i/o operation, the primary i/o process informs the file system (which, in turn, notifies the application process) of the completion.

If the request involves a retryable operation (i.e., a full-sector write and not to append to the end-of-file), the information kept by the file system (i.e, that contained in an i/o request message) is enough the reinitiate the operation. Therefore, in the case of full-sector writes, no checkpointing occurs between the primary and backup i/o processes.

If a failure of the primary i/o process's processor module occurs, the file system and the backup i/o process are notified.

The backup i/o process, when notified of the primary's failure, takes over the primary's duties. The first action that the backup performs is to execute the i/o operation indicated by the latest checkpoint message received from primary i/o process (this occurs regardless of whether or not the operation had been completed by the primary).

When the file system receives notification of the primary's processor module failure, after an operation has been requested but before it has been notified by the i/o process of a successful completion, it reinitiates the operation, this time sending the "i/o request" message (containing the data, sync id, requestor id, and disc address) to the backup i/o process.

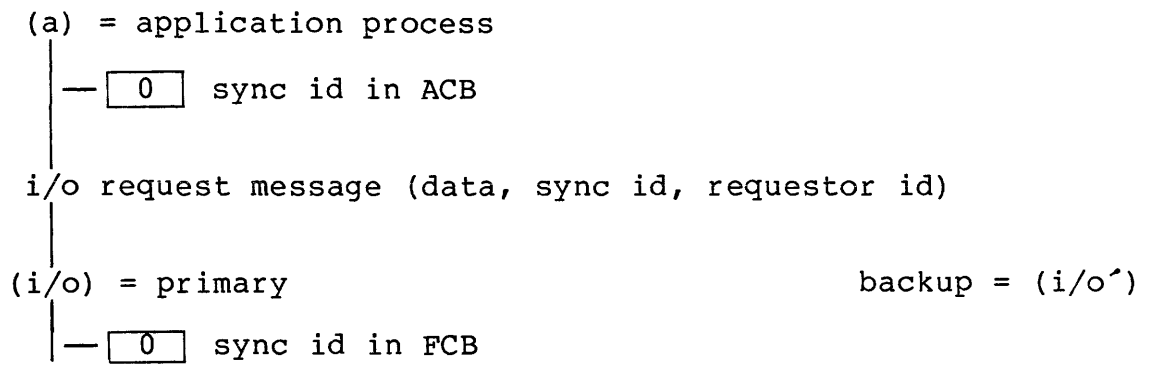
Following a takeover from its primary, the backup i/o process checks the sync id and requestor id in the i/o request message for a match in the list of completed operations. If there is a match, the requested operation has already completed and the backup i/o process returns the associated completion status to the file system (and no other action is taken). If there is no match, the backup i/o process has not performed the operation. The operation is performed in its entirety and the operation's completion status is returned to the file system.

PROCESSOR MODULE FAILURE RECOVERY FOR DISC FILES

The first operation is performed without incident:

```
CALL WRITE(fnum,...);
```

1. The file system sends an "i/o request" message to the primary disc i/o process.



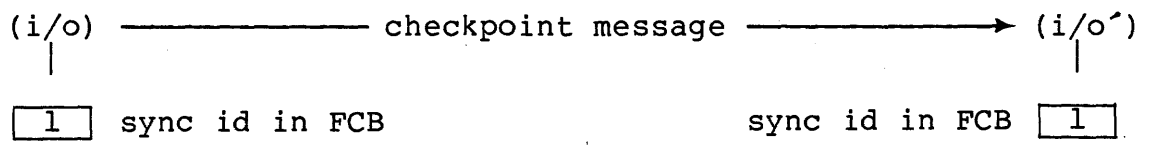
2. In the primary i/o process

* - the sector to be updated is read from disc
- the sector image in memory is updated
- the next sync id (1) is saved.

* performed only if partial-sector write

*3. The state of the operation about to be performed is checkpointed to backup i/o process. The checkpoint message contains:

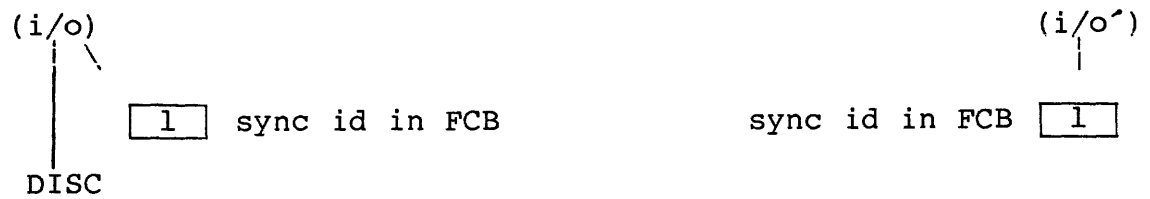
- the requestor id
- the updated sector image
- the next sync id



The backup i/o process

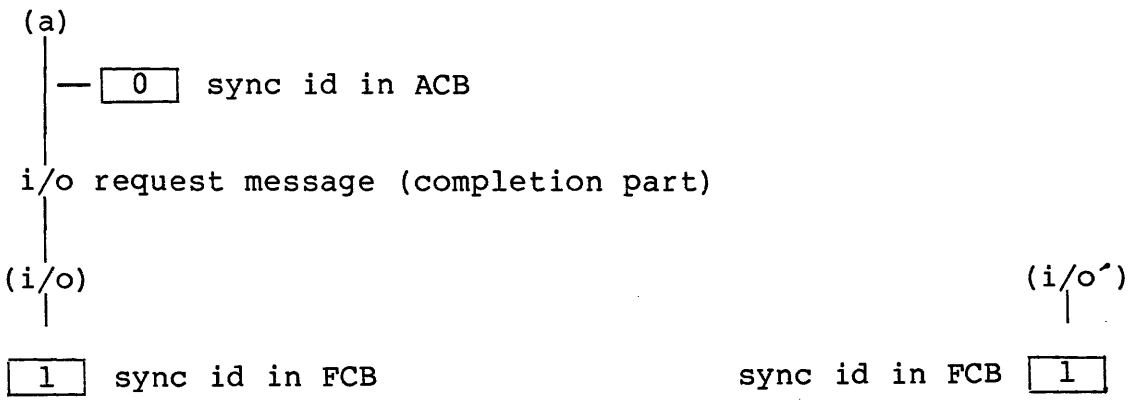
- saves the updated sector image
- saves the next sync id = 1

4. The primary i/o process then writes updated sector image to disc.

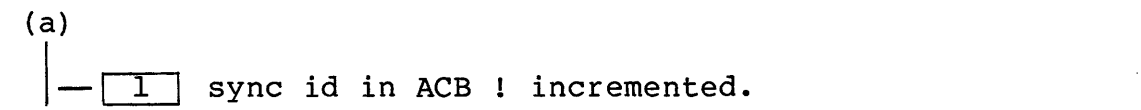


* performed only if partial-sector write

5. The primary i/o process indicates to the application process (i.e., via the file system) that the operation is completed.



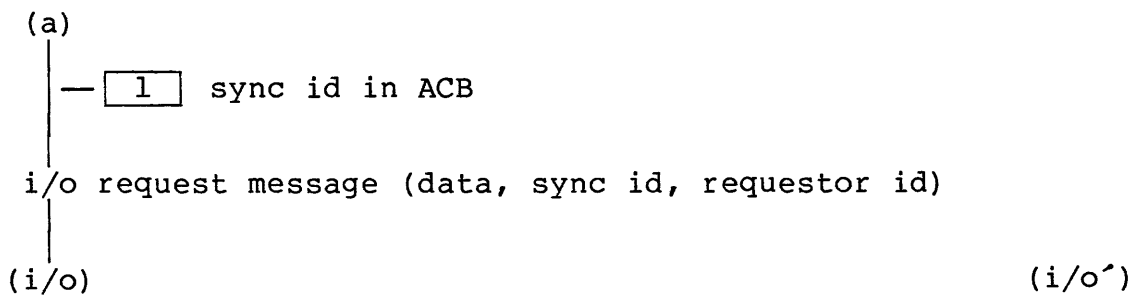
6. The file system increments the "sync id" in the ACB.



The next operation encounters a failure:

CALL WRITE (fnum,...);

1. The file system sends an "i/o request" message to the primary disc i/o process.



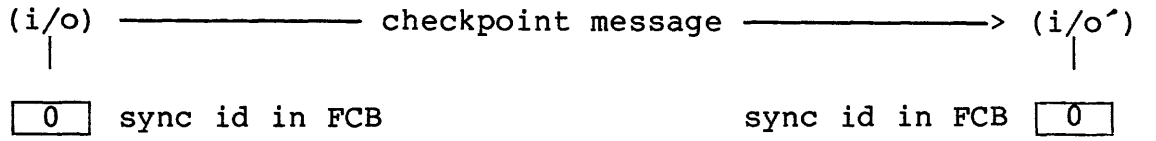
* performed only if partial-sector write

2. In the primary i/o process

* - the sector to be updated is read from disc
- the sector image in memory is updated
- the next sync id (0) is saved.

*3. The state of the operation about to be performed is checkpointed to the backup i/o process. The checkpoint contains:

- the requestor id
- the updated sector image
- the next sync id

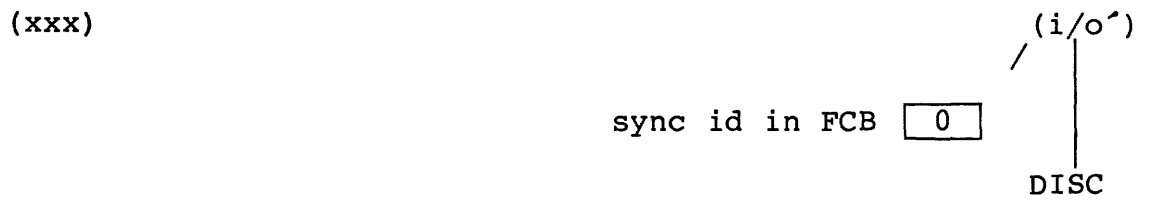


The backup i/o process

- saves the updated sector image
- saves the next sync id = 0

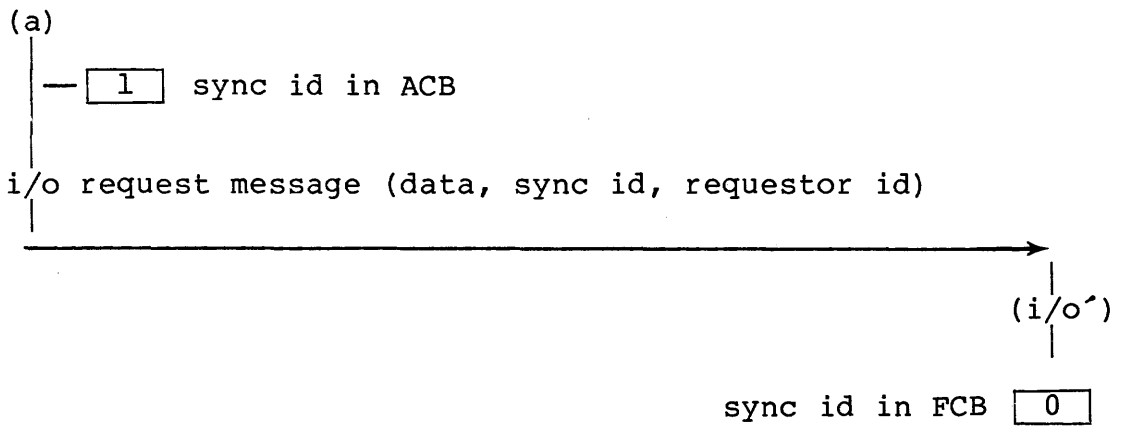
The primary's processor module fails

4. The backup i/o process is notified of the failure. (*) It uses the latest checkpoint from the primary to perform the i/o operation to the disc.

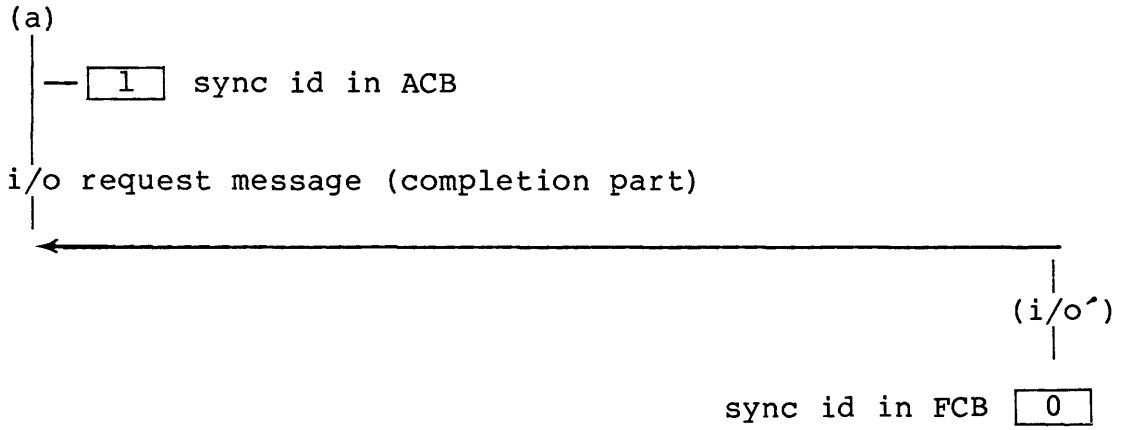


* performed only if partial-sector write

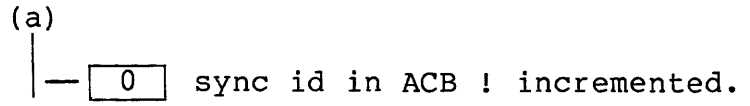
5. The file system, on behalf of the application process, reinitiates the request; this time to the backup process.



6. The backup i/o process compares the requestor id and sync id in the i/o request message with that of operations is has already performed. (*) The backup recognizes that this is a request to perform an operation is has already completed. Therefore, the operation is not performed. Rather, the completion status from the completed operation is returned to the file system.



7. The file system increments the "sync id" in the ACB.



* performed only if partial-sector write

Mirror Volumes

A "mirror" disc volume consists of a pair of physically independent disc devices that are accessed as a single volume; each device is usually controlled through two independent disc controllers. With this configuration, both devices of a mirror volume are controlled by the same i/o process-pair. Each mirror volume is controlled by a separate i/o process-pair. The mirror designation for a volume is indicated to the system at system generation time. The hardware congiguration of a mirror volume is shown in Figure 1-16.

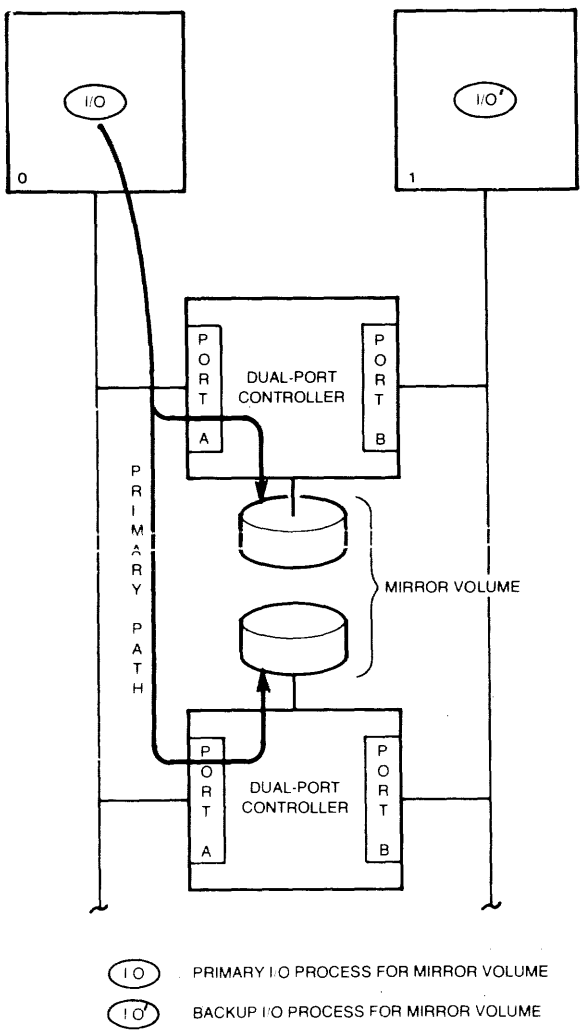


Figure 1-16. Mirror Volume

When a write is performed to a mirror volume, the (primary) i/o process automatically writes the data on the two disc devices comprising the volume. Both devices, when both are operable, are used by the i/o process for reading. If one of the devices becomes inoperable, the i/o process performs all subsequent reading from the operable device.

When an inoperable device is repaired, the information on the previously inoperable pack is brought up-to-date by means of the PUP (Peripheral Utility Program) "REVIVE" command. The REVIVE command copies the information from the operable pack onto the previously inoperable pack in groups of one or more tracks. This copying operation is carried out concurrent with requests to read or update data in files on this volume. (An optional parameter to the REVIVE command specifies a time interval between copying groups of tracks. This permits the revive operation to take place without a significant degradation of system performance.)

Four options are provided to optimize mirror volume performance when both devices comprising a mirror volume are operable. These options, which are specified at system generation time, are:

- for reading, SLAVESEEKS or SPLITSEEKS

  SLAVESEEKS specifies that both devices of a mirror volume are to seek (i.e., perform head positioning) together. The device that is to be used for reading data is selected at random.

  SPLITSEEKS specifies that the device with its head positioned closest to the desired cylinder is the device to be used for reading. The alternate device's head is not repositioned.

- for writing, SERIALWRITES or PARALLELWRITES

  SERIALWRITES specifies that both devices are to seek together when preparing to write. The actual data transfer completes on one device before beginning for the other.

  PARALLELWRITES specifies that both devices are to seek together when preparing to write. Data transfers to both devices occur concurrently. This option is allowed only if each device is controlled by a separate hardware controller.

SECTION 2

ENSCRIBE FILE STRUCTURES

The ENSCRIBE Data Base Record Manager provides these disc file structures:

- Key-Sequenced
- Relative
- Entry-Sequenced
- Unstructured

These four file types fall into two major groups, structured files and unstructured files. Key-sequenced, relative, and entry-sequenced files are structured files, and unstructured files are, of course, unstructured files.

STRUCTURED FILES

This section begins with descriptions of the three stuctured file types. Following the descriptions of key-sequenced, relative, and entry-sequenced files, the section describes basic access concepts for structured files. The section closes with a description of alternate keys, which apply to all three structured file types.

Key-Sequenced Files

A key-sequenced file consists of a set of variable length records. Each record is uniquely identified among other records in a key-sequenced file by the value of its primary key field. Records in a key-sequenced file are logically stored in order of ascending primary key values. The primary key value must be unique and it cannot be changed when updating a record.

A record may vary in length from one byte (1) to the maximum specified for record size when the file was created. The number of bytes allocated for a record is the same as that written when the record was inserted into the file. Each record has a length attribute that is optionally returned when a record is read. A record's length can be changed after the record has been entered (with the restriction that the length cannot exceed the specified record size). Records in a key-sequenced file can be deleted.

A key-sequenced file is physically organized as a tree structure of index blocks and data blocks. Each data block contains 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 which contains 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 new data block is allocated and part of the data from the old block is moved into the new block. In addition, an entry is inserted in the index block to point to the new data block.

When an index block fills up (i.e. there is not enough space in the index block to point to all the data blocks), the block is split into two parts. 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 that this occurs in a file, it is necessary to generate a new level of indices. This is accomplished by allocating a higher level index block which has the low key and pointer to the two lower level index blocks (which in turn point to many data blocks).

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

The user may optionally specify when the file is created that data and/or index records are to be compressed. Compression results in only the significant bytes of the record being stored on the disc. When the record is accessed, it is reconstructed from the significant data and additional information which relates the insignificant data to significant data in another record in the block. Data compression thus reduces storage requirements on the disc at the cost of slightly higher processing time. For sequential processing it can also reduce disc accesses since more records will fit in a block. Data compression can have the additional affect of reducing index levels. When compression is applied to index blocks, it may have the added advantage of reducing the number of index levels so that less reads are necessary to access any data record. This happens because an index block may now point to more data blocks, so there is a proportionate reduction in the number of index blocks.

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

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

The structure of a key-sequenced file is shown in Figure 2-1, on the following page.

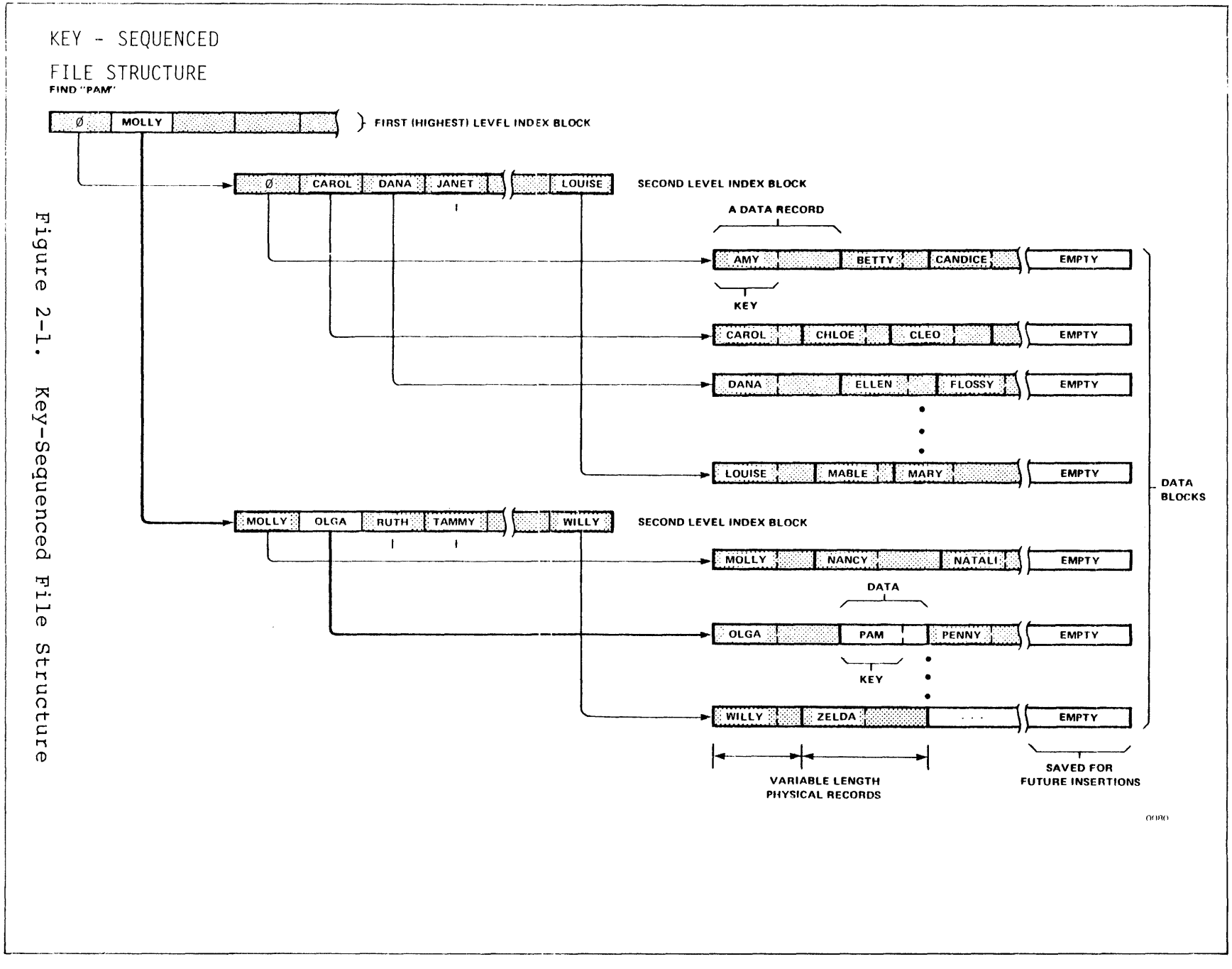


Figure 2-1. Key-Sequenced File Structure

Relative Files

A relative file consists of a set of records. Each record is uniquely identified among other records in a relative file by a record number; a record number denotes an ordinal position in a file. The first record in a relative file is designated by record number zero; succeeding records are designated by ascending record numbers in increments of one. A record occupies a position in a file whether or not the position has been written in.

Each record position is always allocated a fixed amount of storage. However, a record may logically vary in size from byte zero to the maximum specified for record size when the file was created. Each record has a length attribute that is optionally returned when a record is read. A record's length can be changed after the record has been entered (with the restriction that the length cannot exceed the specified record size). Records in a relative file can be deleted.

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

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

The structure of a relative file is shown in Figure 2-2 on the following page.

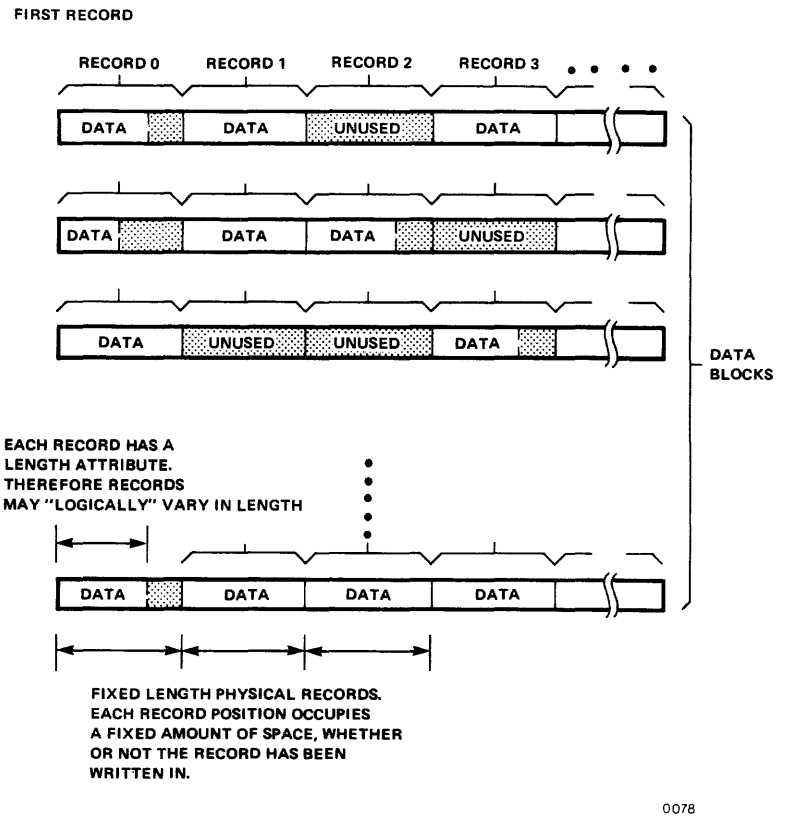


Figure 2-2. Relative File Structure

Relative files are best suited for applications where random access to fixed length records is desired and the record number may function as the key to the file. In the earlier inventory example, it would be possible to make the inventory file a Relative file where the relative record number was equal to the part number. However, this would probably be wasteful of space since part numbering schemes typically leave large gaps in the numbers and this would result in many records allocated but not used. However, an employee file where the relative record number was equal to the employee number would be a good application for a Relative file, since there are typically no large gaps in this kind of file. Data fields in the record could consist of such things as name, address, department, salary, etc.

Entry-Sequenced Files

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

A record may vary in length from zero byte (empty) to the maximum specified for the record size when the file was created. 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 cannot be changed after the record is written into the file. Records in an entry-sequenced file cannot be deleted.

The structure of an entry-sequenced file is shown in Figure 2-3.

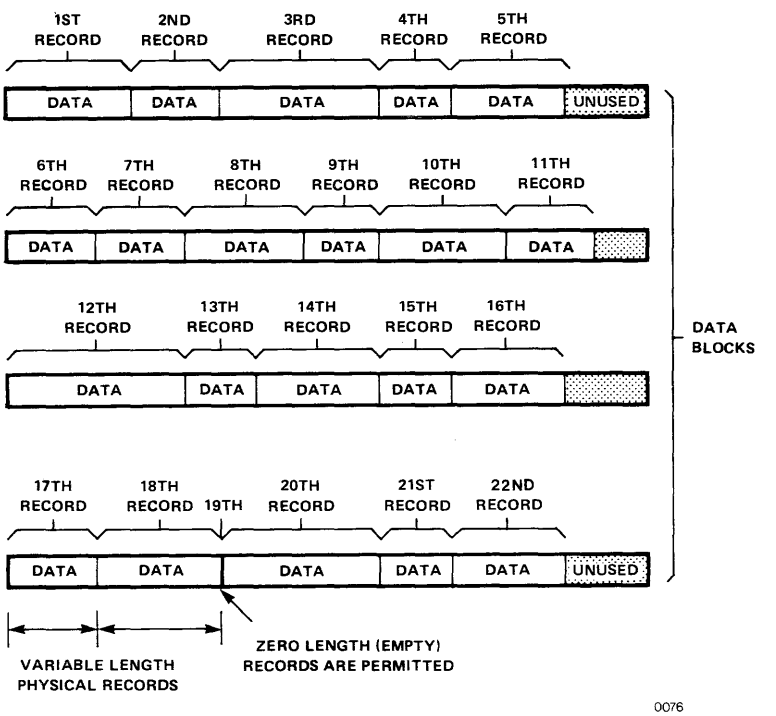


Figure 2-3. Entry-Sequenced File Structure

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

ACCESSING STRUCTURED FILES -- CONCEPTS

This section describes how structured files are accessed. It begins by defining some basic concepts, and then describes how a process can select a subset of a structured file for subsequent access.

Some definitions:

- File — A file is a collection of related records.

- Record — A record is a collection of one or more data items.

- Key — A key is a value that is associated with a record (such as a record number) or contained in a record (such as a byte field) that can be used to locate one record or a subset of records in a file.

- Primary Key — Each record in an ENSCRIBE file is uniquely identified by the value of its "primary" key.

  - for key-sequenced files, this is a byte field within the record and determines where a record is added to a file. The primary key field for a key-sequenced file is defined when the file is created.

  - for relative files, this is a record number.

  - for entry-sequenced files, this is a record address.

- Alternate Key — An alternate key is a byte field within a record that can be used to provide a logically independent access path through a file. The values of an alternate key can be used to identify a subset of records in an access path. A file's alternate key fields are defined when the file is created. Any ENSCRIBE file type can have up to 255 alternate key fields. Alternate key values may or may not be unique.

Current Key Specifier and Current Access Path

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

The current key specifier defines the current access path. The current access path determines the order that records are returned when the file is read sequentially.

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

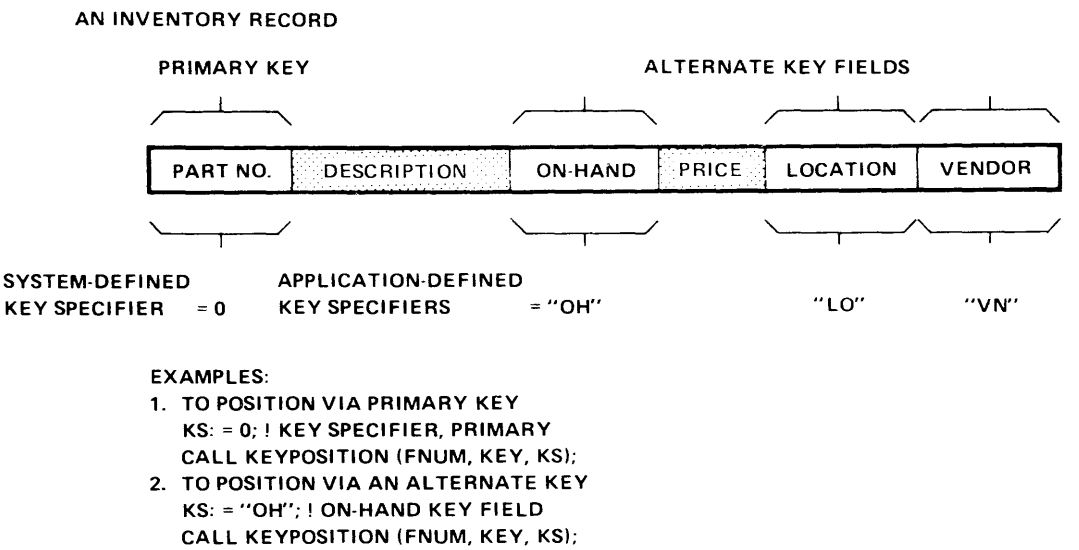


Figure 2-4. Key Fields and Key Specifiers

Current Key Value and Current Position

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

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

The current position determines the record to be locked (by a call to LOCKREC) or accessed (by a call to READ[LOCK], READUPDATE[LOCK], or WRITEUPDATE[UNLOCK]). A record need not exist at the current position. Following a file open, the current position is that of first record in the file as defined by the file's primary key. An example of using KEYPOSITION to position within a key-sequenced file is shown in Figure 2-5 below.

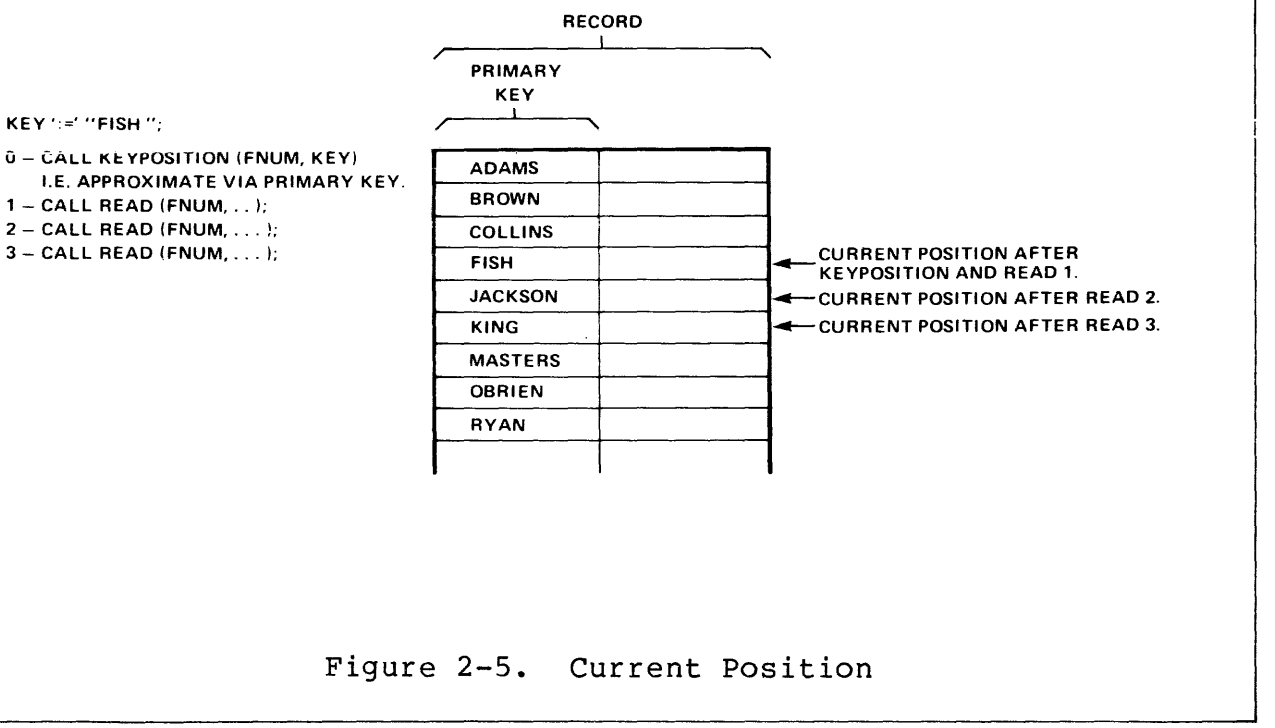


Figure 2-5. Current Position

Positioning Mode and Compare Length

The positioning mode, compare length, and current key value determine the first record accessed and the records comprising a subset of records. Positioning mode and compare length (as well as current key specifier and current key value) are set explicitly by the KEYPOSITION procedure and implicitly by the OPEN and POSITION procedures. There are three positioning modes: approximate, generic, and exact.

APPROXIMATE. Approximate positioning means that 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, or only greater than the current key value. Following approximate positioning, sequential reads to the file return ascending records until the last record in the file is read (an end-of-file indication is then returned). Subsequent to a file open, the positioning mode is set to approximate, the compare length is set to 0.

Sequential reads to a relative file following approximate positioning will skip non-existent records.

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

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

EXACT. Exact positioning means that the only records accessed are those whose key field, as designated by the current key specifier, contains a value of exactly compare length bytes and is equal to the current key value. When the current key no longer matches, an end-of-file indication is returned. Exact positioning on a key field having a unique value accesses at most one record.

Subset

A subset is a related set of records in the current access path. The records comprising a subset are determined by the current key value and positioning mode. A subset may consist of all, part of, or none of the records in a file.

ALTERNATE KEYS

Alternate keys are implemented as follows. For each file having one or more alternate keys, at least one alternate key file exists. Each record in an alternate key file consists of:

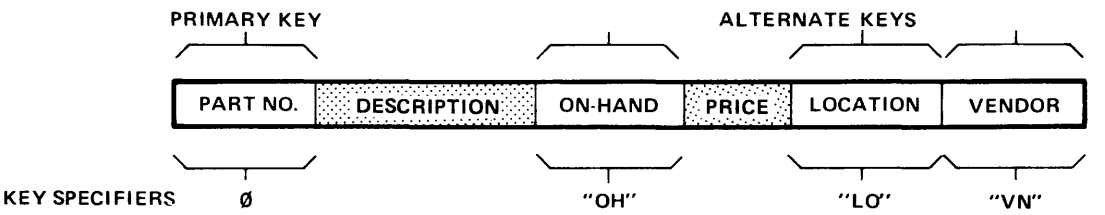
- Two bytes for the <key specifier>
- The alternate key value
- The primary key value of the associated record in the primary file.

The length of an alternate key record is at least

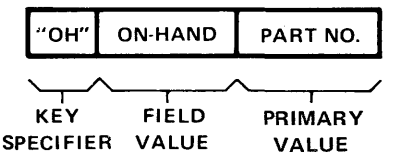
2 + alternate key field length + primary key length

Figure 2-6 on the following page shows how alternate keys are implemented.

RECORD IN PRIMARY FILE



RECORD IN ALTERNATE FILE FOR KEY FIELD "OH"



EXAMPLE

PRIMARY FILE DATA

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	210	5.50	A	BROWN
0310	FRY PAN	19	37.50	D	SMITH
0322	MIXER	12	32.95	D	ACME

ALTERNATE FILE DATA (ALTERNATE FILE IS KEY-SEQUENCED)

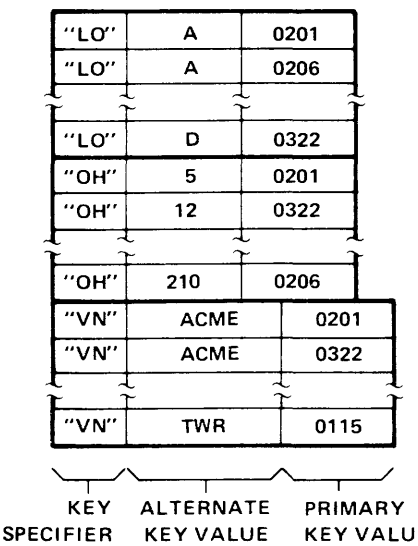


Figure 2-6. Alternate Key Implementation

Alternate Key Attributes

When an alternate key is defined, the following attributes can be assigned.

- Null Value

  An alternate key field can be designated to have a "null value". 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. This has the effect, when reading the file sequentially via an alternate key field having a null value defined, of skipping records containing only the null value in the alternate key field.

- Unique Alternate Key

  An alternate key field can be defined as requiring a unique value. An attempt to insert a record having a duplicate key value in a unique alternate key field is rejected with a file management <error> 10, "record already exists".

- No Automatic Update

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

Alternate Keys in a Key-Sequenced File

An example of alternate key usage in a key-sequenced file would be a file whose records consisted 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 since this is not unique). In order to produce the report consisting of all parts supplied by a given vendor, a "generic position" would be done via the desired vendor. The file is then read sequentially until the vendor name field is not equal to the desired vendor (at which time the system will return an end-of-file indication). The records associated with a given vendor are returned in ascending order of the part number.

Alternate Keys in a Relative File

An example of alternate key usage in a relative file would be a file whose records consisted of employee data. The primary key (i.e., a record number) would be an employee number. An alternate key field would be an employee name.

Alternate Keys in an Entry-Sequenced File

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

Comparison of Structured File Characteristics

COMPARISON OF KEY-SEQUENCED, RELATIVE, AND ENTRY-SEQUENCED FILES

Key-Sequenced	Relative	Entry-Sequenced
Records are ordered by value in primary key field	Records are ordered by relative record number	Records are in the order in which they entered
Access is by primary or alternate key	Access is by record number or alternate key	Access is by record address or alternate key
Space occupied by a record depends on length specified when written	Space occupied by a record is specified when the file is created	Space occupied by a record depends on length specified when written
Free space in block or at end of file is used for adding records	Empty positions in file are used for adding records	Space at end of file is used for adding records
Records can be deleted, shortened, or lengthened	Records can be deleted, shortened, or lengthened	Records cannot be deleted, shortened, or lengthened
Space freed by deleting or shortening a record is reused within its block	Space freed by deleting a record can be reused	A record cannot be deleted, but its space can be used for another record of the same size

UNSTRUCTURED FILES

This section begins with a list of unstructured disc file characteristics and then briefly describes how an application process can position within an unstructured file.

Unstructured File Characteristics

Unstructured files have the following characteristics:

- It is the application's responsibility to determine optimum record sizes and to block records in an efficient manner.
- Files must be created first, then opened for access.
- Disc space is allocated by the file system in file extents as required. A file may have as many as 16 extents. The first extent is designated the "primary" extent and may differ in size from the remaining "secondary" extents.
- An application process can initially allocate one or more extents in an open file via a call to CONTROL, <operation> = 21. This CONTROL <operation> can also be used to deallocate unused extents.
- File names for permanent files are of the form

  ```
  <file name[0:3]>  is $<volume name><blank fill>
  <file name[4:7]>  is <subvol name><blank fill>
  <file name[8:11]> is <disc file name><blank fill>
  ```

- File names for temporary files are of the form

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

- For network file names, <file name[4:11]> is the same as for local file names but

  ```
  <file name[0:3]>  is \<sys num><volume name><blank fill>
  ```

- Data in an unstructured disc file is referenced by a relative byte address. Three file pointers indicate the current address just accessed, the next address to be accessed, and the end-of-file address
- Data is physically located on disc in 512-byte sectors.

- If a data transfer is between 3585 and 4095 bytes inclusive, the transfer may fail if the current record pointer is positioned such that the transfer spans more than eight sectors. The condition which must be met for the transfer to be possible is:

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

- An application process can purge all of the data from a file, without deleting the file, by use of the CONTROL procedure, "purge data" operation.

- File locking procedures are provided to coordinate access to a shareable file

- Error Recovery

  Parity and overrun errors are retried automatically by the file system. Therefore, if one of these errors is reported back to the application process, the file is no longer accessible.

  Path errors are retried automatically by the file system if the file is open with a "synchronization depth" greater than zero. An error return of a path error in this case then indicates that the file is no longer accessible.

- Maximum number of files on a <volume> is a function of system configuration

- <device type> is 3

Relative Byte Addressing and File Pointers

Data in an unstructured disc 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 a file; the first byte in a file is at rba zero.

Three file pointers are associated with each open disc file:

1. A next-record pointer containing the relative byte address of the location where the next disc transfer, due to a READ or WRITE, begins.

2. A current-record pointer containing the relative byte address of the location just read or written and is the address where a disc transfer due to a READUPDATE or WRITEUPDATE begins.

3. An end-of-file pointer containing the relative byte address of the next even numbered byte after the last significant data byte in a file. The end-of-file pointer is incremented automatically when data is appended to the end-of-file (WRITE). It can be set explicitly by calls to the POSITION and CONTROL procedures.

Separate next-record and current-record pointers are associated with each open of a disc file so that if the same file is open several times simultaneously, each open provides a logically separate access. The next-record and current-record pointers reside in the file's Access Control Block in the application process environment.

A single end-of-file pointer, however, is associated with all opens of a given disc file. This permits data to be appended to the end-of-file by several different accessors. The end-of-file pointer resides in the file's File Control Block in the disc i/o process environment. A file's end-of-file pointer value is copied from the file label on disc when the file is opened and is not already open; the end-of-file pointer value in the file label is updated when any CONTROL operation to the file is performed, when a file extent is allocated for the file, and when the file is closed and there are no other opens of the file.

A summary of file pointer action is given in Table 2-1 on the following page.

Table 2-1. File Pointer Action

```
CREATE

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

OPEN (first)

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

OPEN (any)

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

READ

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

WRITE

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

READUPDATE

   file pointers are unchanged

WRITEUPDATE

   file pointers are unchanged

CONTROL (write end-of-file)

   end-of-file pointer := next-record pointer;
   file label end-of-file pointer := end-of-file pointer;
```

→

Table 2-1. File Pointer Action (cont'd)

CONTROL (purge data)

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

CONTROL (allocate/deallocate extents)

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

POSITION

current-record pointer := next-record pointer := <rba>;

CLOSE (last)

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

where

<count> is the specified transfer count rounded-up to an even number

SECTION 3

ENSCRIBE FILE MANAGEMENT PROCEDURES

An application process accesses ENSCRIBE disc files through calls to ENSCRIBE file management procedures. This section contains detailed syntax descriptions of all the ENSCRIBE file management procedures.

The section begins with a brief summary of all of the ENSCRIBE calls. Then characteristics common to all of the calls are described. The section continues with a full syntax description of disc file names, both for single system and network applications. Following the file name description, syntax descriptions of all the ENSCRIBE file management calls are listed in alphabetical order.

FILE MANAGEMENT CALL SUMMARY

A functional summary of all the ENSCRIBE file management calls is given in Table 3-1 below.

Table 3-1. File Management Call Summary

AWAITIO	waits for completion of an outstanding i/o operation pending on an open file
CANCELREQ	cancels the oldest outstanding operation, optionally identified by a <tag>, on an open file.
CLOSE	stops access to an open file and purges a temporary disc file
CONTROL	executes device dependent operations to an open file
CREATE	creates a new disc file (permanent or temporary)

Table 3-1. File Management Call Summary (cont.)

DEVICEINFO	provides the device type and physical record size for a file (open or closed)
EDITREADINIT	prepares a control block for subsequent calls to EDITREAD
EDITREAD	reads a line of text (logical record) from an EDIT file
FILEERROR	is used to determine whether or not a failed call should be retried
FILEINFO	provides error information and characteristics about an open file
FILERECINFO	provides characteristic information about an open Enscribe disc file
FNAMECOLLAPSE	collapses an internal file name to external form
FNAMECOMPARE	compares two internal file names to determine whether the two names refer to the same file or device
FNAMEEXPAND	expands an external file name to internal form
GETDEVNAME	returns the $<device name> or $<volume name> associated with a logical device number if such a device exists; otherwise the name of the next higher logical device number is returned
KEYPOSITION	sets the current key value and current key specifier for an open Enscribe disc file provides the <system number> corresponding to a <system name>
LOCKFILE	locks an open disc file, making the file inaccessible to other accessors
LOCKREC	locks a record in an open disc file so that other processes cannot access the record
NEXTFILENAME	returns the next disc file name in alphabetical sequence following the designated file name

→

Table 3-1. File Management Call Summary (cont.)

OPEN	establishes communication with a file
POSITION	set the current (primary) key for an open relative or entry-sequenced file
PURGE	purges a closed disc file from the system
READ	is used when sequentially reading an open file. It returns the record indicated by the value of current key
READLOCK	is the same as READ but first locks the record before reading it
READUPDATE	is used to randomly read an open file. It returns the record indicated by the current key value
READUPDATELOCK	is the same as READUPDATE except that it first locks the record before reading it
REFRESH	is used to write the information contained in File Control Blocks (FCBs) in main-memory, such as the end-of-file pointer, to the associated physical disc volume
RENAME	renames an open disc file and makes a temporary disc file permanent which message is being replied to
REPOSITION	restores the disc file positioning information saved with previous SAVEPOSITION
SAVEPOSITION	saves the current disc file position information; a later call to REPOSITION restores the saved position
SETMODE	sets device-dependent functions in an open file
SETMODENOWAIT	sets device-dependent functions in a no-wait manner for an open file.

→

Table 3-1. File Management Call Summary (cont.)

UNLOCKFILE	unlocks an open disc file currently locked by the caller Additionally, a call to UNLOCKFILE unlocks any records in the designated file that are currently locked by the caller
UNLOCKREC	unlocks a record currently locked by the caller so that other processes can access the record
WRITE	inserts (adds) a new record into an open disc file location read by the last call to READ or READUPDATE
WRITEUPDATE	replaces (updates) or deletes data in the existing record indicated by an open file's current key value
WRITEUPDATEUNLOCK	is the same as WRITEUPDATE except that the record is unlocked after it is updated or deleted

CHARACTERISTICS OF ENSCRIBE CALLS

This section describes features common to all ENSCRIBE file management calls.

Completion

If a file is open with no-wait i/o specified, the following calls must be completed by a corresponding call to AWAITIO:

CONTROL, LOCKFILE, LOCKREC, READ, READLOCK, READUPDATE, READUPDATELOCK, UNLOCKFILE, UNLOCKREC, WRITE, WRITEUPDATEUNLOCK, and SETMODENOWAIT.

If a file is open with no-wait i/o specified, the following calls are rejected with a file management error 27 if there are any outstanding (i.e., uncompleted) operations pending:

KEYPOSITION, POSITION, RENAME, REPOSITION, SETMODE, and SETMODENOWAIT.

Regardless of whether the file was opened with wait or no-wait i/o specified, a return from the following calls indicates a completion:

CANCELREQ, CLOSE, CREATE, DEVICEINFO, FILEINFO, FILERECINFO KEYPOSITION, NEXTFILENAME, OPEN (unless flag <8> is set to 1), POSITION, PURGE, RENAME, and SETMODE.

<file number> Parameters

All file management procedures except

DEVICEINFO, CREATE, OPEN, NEXTFILENAME, REFRESH, and PURGE

use the <file number> returned from the OPEN procedure to identify which file the call references. The DEVICEINFO, CREATE, OPEN, and PURGE procedures reference the file by its <file name>; the LASTRECEIVE and REPLY procedures always reference the $RECEIVE file (i.e., interprocess communication). For every procedure that has a <file number> parameter, except OPEN and AWAITIO, the file number is an INT:value parameter.

<tag> Parameters

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

<buffer> Parameter

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

<transfer count> Parameter

The transfer count parameter of file management procedures always refers to the number of BYTES to be transferred. The number of bytes that can be transferred in a single operation with an Enscribe disc file is in the range of {0:4096}. This figure is the file system/hardware maximum. The actual maximum transfer count may be less than 4096 due to the amount of buffer space assigned to the disc at system generation time (SYSGEN). (The amount buffer space configured for a disc volume can be obtained via the DEVICEINFO procedure.)

Condition Codes

All file management procedures return a condition code indicating the outcome of the operation. THE CONDITION CODE SHOULD ALWAYS BE CHECKED FOLLOWING A CALL TO A FILE SYSTEM PROCEDURE and should be checked before an arithmetic operation or a store into a variable is performed. Generally, the condition codes have the following meanings:

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

Errors

Associated with each call completion is an error number. The error numbers into fall into three major categories. As shown below, the setting of the condition code indicates the category of the error associated with a completed call.

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

The error number associated with an operation on an open file can be obtained by calling the FILEINFO procedure and passing the <file number> of the file in error:

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

The error number associated with an unopen file or a file open failure can be obtained by passing the <file number> as -1 to the FILEINFO procedure:

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

Note: the OPEN procedure returns -1 to <file number> if the open fails.

Error recovery is described in Section 4, Enscribe File Access.

A complete list of the error numbers and their meanings is given in Appendix D, File Error Summary.

Access Mode and Security Checking

READ ACCESS. The disc file must be open with read or read/write access for the following calls to be successful (otherwise the call will be rejected with a file management <error> 49, "access violation"):

READ, READLOCK, READUPDATE, and READUPDATELOCK.

WRITE ACCESS. The disc file must be open with write or read/write access for the following calls to be successful (otherwise the call will be rejected with a file management <error> 49, "access violation"):

CONTROL, WRITE, WRITEUPDATE, and WRITEUPDATEUNLOCK.

PURGE ACCESS. The caller must have purge access to a disc file for the following calls to be successful (otherwise the call will be rejected with a file management <error> 48, "security violation"):

PURGE and RENAME.

Current State Indicators

For each file management procedure, changes to the current state indicators are listed. The current state indicators are:

- current position.
- positioning mode.
- compare length.
- current primary key value.

EXTERNAL DECLARATIONS

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

```
<global declarations>

?SOURCE $SYSTEM.SYSTEM.EXTDECS ( <ext proc name> , ... )

<procedure declarations>
```

Each external procedure that is referenced in the program should be specified in the SOURCE command.

For example:

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

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

FILE NAMES

File names are used when creating new disc files, purging old disc files, and renaming disc files.

There are two forms of file name - "external" and "internal". The external form is used when entering file names into the system from the outside world (e.g., by a user to specify a file name to the Command Interpreter). The external form is described in the GUARDIAN Programming Manual. The internal form is used within the system when passing file names between application processes and the operating system. This section describes the internal form.

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

The internal form of disc file names is:

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

where

   to access permanent disc files, use

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

   to access temporary disc files, use

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

Permanent Disc File Names

Permanent disc file names are of the form:

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

Each of these three components of a disc file name is described below.

- <volume name>

<volume names> identify disc packs (each pack in the system has a <volume name>). They are assigned at system generation time and when new disc packs are introduced into the system. A <volume name> must be preceded by a dollar sign "$" and consists of a maximum of seven alphanumeric characters; the first character must be alphabetical.

- <subvol name>

This name identifies a subset of disc files. <subvol names> are assigned programmatically when disc files are created. A <subvol name> consists of a maximum of eight alphanumeric characters; the first character must be alphabetical.

- <disc file name>

This name identifies a particular disc file. <disc file names> are assigned programmatically when disc files are created. A <disc file name> consists of a maximum of eight alphanumeric characters; the first character must be alphabetical.

Temporary File Names

Temporary file names identify temporary disc files. <temporary file names> are assigned by the file management CREATE procedure when temporary files are created. A <temporary file name> consists of a number sign "#" followed by four numerical characters.

File Name Examples

Permanent disc file:

```
INT .fname[0:11] := "$STORE1 ACCT1   MYFILE  ";
```

temporary disc file:

```
INT .fname[0:11] := ["$STORE1 ", 8 * ["  "]];
```

only the <volume name> is supplied. The <temporary file name> is returned from CREATE.

```
CALL CREATE(fname);
```

Network File Names

File names can optionally include a <system number> that identifies a file as belonging to a particular system on a network. (See the EXPAND User's Manual for information regarding networks of Tandem systems.)

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

```
<network file name>  ! 12 words, blank filled

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

where

   <system number>

      is an integer between 0 and 254 that designates a
      particular system.  The assignment of system numbers is
      made at system generation (SYSGEN) time

   <volume name>

      consists of at most six alphanumeric characters, the first
      of which must be alphabetic
```

Note that names of disc volumes and other devices, when embedded within a network file name, are limited to having six characters, and do NOT begin with a dollar sign. Similar restrictions apply to the network form of <process id>, as follows.

> Note that <process name> in words 1 and 2 can contain at most four alphanumeric characters (the first one must be alphabetic, as usual) and does NOT include the initial dollar sign "$".

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

Conversion between internal and external forms of network file names is accomplished by the procedures FNAMEEXPAND and FNAMECOLLAPSE.

AWAITIO

The AWAITIO procedure is used to complete a previously initiated "no-wait" i/o operation. AWAITIO can be used to -

- Wait for a completion with a particular file. Application process execution is suspended until the completion occurs. A timeout is considered to be a completion in this case.
- Wait for a completion with any file or a timeout to occur. A timeout is not considered to be completion in this case.
- Check for a completion with a particular file. The call to AWAITIO immediately returns to the application process regardless of whether there is a completion or not. (If there is no completion, an error indication is returned.)
- Check for a completion with any file.

If AWAITIO is used to wait for a completion, a time limit can be specified as to maximum time allotted to completing the waited-for operation.

The call to the AWAITIO procedure is:

```
CALL AWAITIO (     <file number>
              , [ <buffer address>    ]
              , [ <count transferred> ]
              , [ <tag>               ]
              , [ <time limit>        ]  )
```

where

<file number>, INT:ref:1, passed, returned

if a particular <file number> is passed, AWAITIO applies to that file. The specific action depends on the value of the <time limit> parameter. If <time limit> is a non-zero value, the application process is suspended until a completion occurs or the <time limit> expires. If passed as 0D, a completion check is made.

If <file number> is passed as -1, the call to AWAITIO applies to the oldest outstanding operation pending on any file. The specific action depends on the value of the <time limit> parameter. If <time limit> is a non-zero value, the application process is suspended until a completion occurs or the <time limit> expires. If <time limit> is passed as 0D, a completion check is made. In either case, if an operation completed, AWAITIO returns to <file number> the file number associated with the completion.

<buffer address>, INT:ref:1, passed

if present, returns the address of the <buffer> specified when the operation was initiated. Note that if the actual parameter is to be used as an address pointer to the returned data and has been declared in the form "INT .<buffer address>", it should be passed to AWAITIO in the form "@<buffer address>".

<count transferred>, INT:ref:1, returned

if present, returns the count of the number of bytes transferred because of the associated operation.

→

<tag>, INT(32):ref:1, returned

if present, returns the application-defined tag that was stored by the system when the i/o operation associated with this completion was initiated.

<time limit>, INT(32):value, passed

if present, indicates whether the process wants to wait for completion or check for completion:

If <time limit> <> 0D then a wait-for-completion is specified. <time limit> then specifies the maximum time (in .01 second units) that the application process is willing to wait (i.e., be suspended) for completion of a waited-for operation. Specifying a <time limit> value of -1D implies a willingness to wait forever.

If <time limit> = 0D then a check-for-completion is specified. AWAITIO immediately returns to the caller regardless of whether or not an i/o completion occurred.

If <time limit> is omitted, then a willingness to wait forever is specified.

condition code settings:

```
<  (CCL) indicates that an error occurred (call FILEINFO)
=  (CCE) indicates that an i/o completed
>  (CCG) indicates that a warning occurred (call FILEINFO)
```

example

```
CALL AWAITIO ( file^num );
IF < THEN ...                    ! error occurred.
```

Considerations

- Completing No-wait I/O Calls

  Each "no-wait" operation initiated must be completed with a corresponding call to AWAITIO.

  - If AWAITIO is used to wait for completion (i.e., <time limit> <> 0D) and a particular file is specified (i.e., <file number> <> -1), then completing AWAITIO for any reason is considered a completion.

  - If AWAITIO is used to check for completion (<time limit> = 0D) or used to wait on any file (<file number> = - 1), completing AWAITIO does not necessarily indicate a completion. If an error indication is returned and a subsequent call to FILEINFO returns <error> = 40 (i.e., a timeout), then the operation is considered incomplete (AWAITIO must be called again). Any indication other than <error> = 40 (i.e., CCE, CCG, CCL and <error> <> 40) indicates a completion.

- Order of I/O Completion With SETMODE 30 Set

  Specifying SETMODE 30 allows no-wait i/o operations to complete in any order. When initiating an i/o operation, an application process employing this option can use the <tag> parameter to keep track of multiple operations associated with an open of a file.

- Order of I/O Completion Without SETMODE 30 Set

  If SETMODE 30 has not been set, the oldest outstanding i/o operation is always completed first. Therefore, AWAITIO always completes i/o operations associated with the particular open of a file in the same order as initiated.

- Error Handling

  If an error indication is returned (i.e., condition code is CCL or CCG), the <file number> that is returned by AWAITIO can be passed to the FILEINFO procedure to determine the cause of the error. If <file number> = -1 (i.e., any file) is passed to AWAITIO and an error occurs on a particular file, AWAITIO returns, in <file number>, the actual file number associated with the error.

- Error 26: No Outstanding No-wait I/O Calls

  If AWAITIO is called and a corresponding "no-wait" operation has not been initiated, an error indication is returned (CCL) and a subsequent call to FILEINFO returns <error> = 26 (no outstanding operation).

- WRITE Buffers

  The contents of a buffer being written should not be altered between a no-wait i/o initiation (e.g., call to WRITE) and the corresponding no-wait i/o completion (i.e., call to AWAITIO). If the buffer is altered, application error recovery can become difficult, if not impossible.

- AWAITIO Completion

  The way in which AWAITIO completes depends on whether the <file number> parameter specifies a particular file or any file, and on the value of <time limit> passed with the call. The action taken by AWAITIO for each combination of <file number> and <time limit> is summarized in Table 3-1, on the following page. (Note: Table 3-1 assumes SETMODE 30 has been set.)

- AWAITIO Operation

  Figure 3-1 illustrates the operation of the AWAITION procedure.

Table 3-1. AWAITIO Action

	⟨time limit⟩ = 0	⟨time limit⟩ ≠ 0
Particular File ⟨fn⟩ = ⟨file num⟩	**CHECK** for any ⟨file num⟩ I/O completion **COMPLETION** file number returned in ⟨fn⟩ ; Tag of completed call returned in ⟨tag⟩ **NO COMPLETION** CCL (error 40) returned; file number returned in ⟨fn⟩ ; No I/O operation is canceled.	**WAIT** for any ⟨file num⟩ I/O completion **COMPLETION** file number returned in ⟨fn⟩ ; Tag of completed call returned in ⟨tag⟩ . **NO COMPLETION** CCL (error 40) returned; file number returned in ⟨fn⟩ ; ***Oldest*** ⟨file num⟩ I/O operation canceled; Tag of canceled call returned in ⟨tag⟩ .
Any File ⟨fn⟩ = −1	**CHECK** for any I/O completion on any open file **COMPLETION** File number of completed call returned in ⟨fn⟩ ; Tag of completed call returned in ⟨tag⟩ . **NO COMPLETION** CCL (error 40) returned; −1 returned in ⟨fn⟩ ; No I/O operation is canceled.	**WAIT** for any I/O completion on any open file **COMPLETION** File number of completed call returned in ⟨fn⟩ ; Tag of completed call returned in ⟨tag⟩ . **NO COMPLETION** CCL (error 40) returned; −1 returned in ⟨fn⟩ ; No I/O operation is canceled.

Notes: ⟨fn⟩ = ⟨file number⟩
SETMODE 30 Set

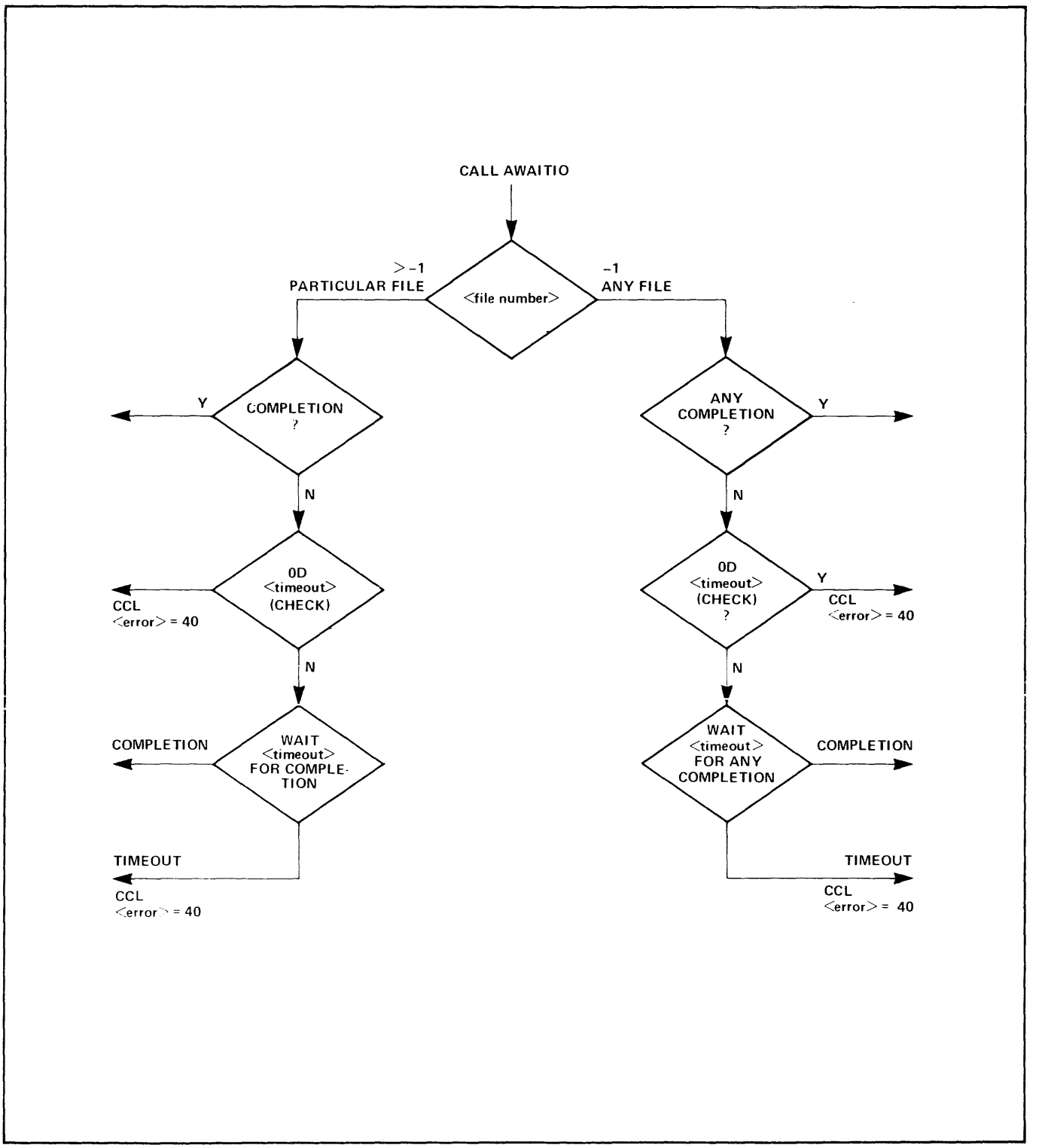


Figure 3-1. AWAITIO OPERATION

CANCEL

The CANCEL procedure is used to cancel the oldest outstanding operation on a no-wait file. (Note: A specific call, identified with a <tag> parameter, can be canceled with a call to CANCELREQ.)

The call to the CANCEL procedure is:

```
CALL CANCEL ( <file number> )

where

   <file number>, INT:value,                                  passed

     identifies the file whose oldest outstanding operation is to
     be canceled.


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the operation was cancelled
   >  (CCG) is not returned by CANCEL


example

   CALL CANCEL ( some^file );
   IF < THEN ..... ! no operation outstanding.
```

CANCELREQ

The CANCELREQ procedure is used to cancel an outstanding operation identified by <file number> and <tag> on a no-wait file.

The call to the CANCELREQ procedure is:

```
CALL CANCELREQ   (    <file number>
                  , [ <tag>         ] )
```

where

<file number>, INT:value, passed

identifies the file whose outstanding operation is to be canceled.

<tag>, INT(32):value, passed

if present, identifies the operation to be canceled. <tag> is the application-defined tag that is stored by the system when the i/o operation is initiated.

condition code settings:

<	(CCL) indicates that an error occurred (call FILEINFO)
=	(CCE) indicates that the operation was canceled
>	(CCG) is not returned by CANCELREQ

example

```
CALL CANCELREQ ( some^file,14D);
IF = THEN ..... ! operation 14 of some^file canceled.
```

Considerations

o Using the <tag> Parameter

If the <tag> parameter is provided, the oldest outstanding operation with that tag value is canceled. If <tag> is not provided, the oldest outstanding operation for <file number> is canceled.

CLOSE

The CLOSE procedure is used to terminate access to an open file.

When a permanent disc file is closed, if it is not open concurrently, the file label on disc is updated with pertinent information from the main-memory resident File Control Block and the space in use by the FCB is returned to a system main-memory space pool. When a temporary disc file is closed, if it is not open concurrently, its name is deleted from the volume's directory and any space that had been allocated to the file is made available for other files.

For any file close, the space allocated to the Access Control Block is returned to the system.

The call to the CLOSE procedure is:

```
CALL CLOSE ( <file number> )

where

   <file number>, INT:value,                              passed

     identifies the file to be closed.


condition code settings:

   <  (CCL) indicates that the file was not open
   =  (CCE) indicates that the CLOSE was successful
   >  (CCG) is not returned by CLOSE


example

   CALL CLOSE ( fnum );
```

Considerations

- Closing a No-wait File

  If a CLOSE is executed to a no-wait file with outstanding operations pending, any uncompleted operations are canceled. There is no indication as to whether the operation completed or not.

CONTROL

The CONTROL procedure is used to perform device-dependent i/o operations.

If the CONTROL procedure is used to initiate an operation with the file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the CONTROL procedure is:

```
CALL CONTROL (    <file number>
              ,   <operation>
              ,   <parameter>
              , [ <tag>         ] )
```

where

<file number>, INT:value, passed

identifies the file that is to execute the control operation.

<operation>, INT:value, passed

is defined by device in the table that follows.

<parameter>, INT:value, passed

is also defined in the table that follows.

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the control operation completes.

condition code settings:

<	(CCL)	indicates that an error occurred (call FILEINFO)
=	(CCE)	indicates that the CONTROL was successful
>	(CCG)	is not returned by CONTROL

→

```
example

   CALL CONTROL ( printer, form^control, vfu^channel );
   IF < THEN ....                    ! error occurred.
```

The control operations that apply to disc files are listed in Table 3-3 below.

Table 3-3. CONTROL OPERATIONS

<operation>	
2 = write end-of-file (unstructured files only. Write access is required)	
	<parameter> = number of records {0:255}
20 = disc, purge data (write access is required)	
	<parameter> = none
21 = disc, allocate/deallocate extents (write access is required)	
	<parameter> = 0 = deallocate all extents past the end-of-file extent
	1:16 = number of extents to allocate for a key-sequenced file
	1:<total extents> = total number of extents to allocate for entry-sequenced, relative and unstructured files
	where
	<total extents> = 16 * <number of partitions>

Considerations

- Writing EOF to an Unstructured File

  A write end-of-file to an unstructured disc file sets the end-of-file pointer to the relative byte address indicated by the setting of the next-record pointer and writes the new end-of-file setting in the file label on disc. Specifically:

  ```
  end-of-file pointer := next-record pointer;
  ```

- Error 73: File is Locked

  If a control operation is attempted for a file that is locked through a <file number> other than the <file number> specified with the call to CONTROL, the call to CONTROL is rejected with an error 73: file is locked.

- Allocating Extents for Partitioned Key-Sequenced Files

  To allocate extents for partitioned key-sequenced files, each partition must be opened separately and a CONTROL 21 issued for the individual partition.

CREATE

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

The call to the CREATE procedure is:

```
CALL CREATE (    <file name>
             , [ <primary extent size>   ]
             , [ <file code>             ]
             , [ <secondary extent size> ]
             , [ <file type>             ]
             , [ <record length>         ]
             , [ <data block length>     ]
             , [ <key-sequenced params>  ]
             , [ <alternate key params>  ]
             , [ <partition params>      ]  )
```

where

<file name>, INT:ref, passed, [returned]

is an array providing the name of the disc file to be created in either of the following forms:

permanent disc files are created by specifying

```
<file name[0:3]>  is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
<file name[4:7]>  is <subvol name><blank fill>
<file name[8:11]> is <disc file name><blank fill>
```

temporary disc files are created by specifying

```
<file name[0:11]> is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
```

when CREATE completes, a <temporary file name> is returned in <file name[4:7]>. The temporary file can then be opened by passing <file name> to OPEN.

<primary extent size>, INT:value, passed

if present, is the size of the primary extent in 2048-byte units. The maximum <primary extent size> is 65535 (134,215,680 bytes). If omitted, a primary extent size of one (2048 bytes) is assigned.

<file code>, INT:value, passed

if present, is an application-defined file identification code (file codes 100 - 999 are reserved for use by Tandem Computers, Inc.). If omitted, a file code of zero is assigned.

<secondary extent size>, INT:value, passed

if present, is the size of the secondary extents in 2048-byte units (a file may have up to 15 secondary extents allocated). The maximum <secondary extent size> is 65535 (134,215,680 bytes). If omitted, the size of the primary extent is used for the secondary extent size.

<file type>, INT:value, passed

if present, specifies the type of the file to be created.

<file type>.<13:15> specifies the file structure:

0 = unstructured (default)
1 = relative
2 = entry-sequenced
3 = key-sequenced

<file type>.<12> 1 = specifies 'ODDUNSTR' for unstructured files. See Section 5, File Creation, for details.

<file type>.<12> 1 = specifies data compression for key-sequenced files. See Section 5, File Creation, for details.

<file type>.<11> 1 = specifies index compression for key-sequenced files. See Section 5, File Creation, for details.

<file type>.<10> 1 = File Label is written to disc each time the end-of-file is advanced. The effect of setting this parameter is the same as calling REFRESH after every operation that advances the end-of-file.

<file type>.<3:9> must be zero.

<file type>.<2> 1 = for systems with the Transaction Monitoring Facility (TMF), specifies this file is an audited file; for systems without TMF, must be zero.

<file type>.<0:1> must be zero

If <file type> is omitted, an unstructured file is created.

<record length>, INT:value, passed

if present, is the maximum length of the logical record in bytes. For structured files, the maximum record length is determined by the data block size. With a data block size of 4096, the maximum record length for entry-sequenced and relative files is 4072 bytes. With the same maximum data block size of 4096, the maximum record length for a key-sequenced file is 2035. For unstructured files, the maximum record length is 4096. If omitted, 80 is used for the <record length>.

<data block length>, INT:value, passed

for structured files, if present, is the length in bytes of each block of records in the file. <data block length> must be a multiple of 512 and can not be greater than 4096. <data block length> must be at least <record length> + 24. For a key-sequenced file <data block length> must be at least 2 * <record length> + 26. If omitted, 1024 is used for the <data block length>. Regardless of the specified record length and data block size, the maxixum number of records that can be stored in a data block is 511.

→

<key-sequenced params>, INT:ref, passed

is a three-word array containing parameters that describe this file. This parameter is required for key-sequenced files and may be omitted for other file types. The format for this array is shown in the "<key sequenced params> ARRAY" table which follows

<alternate key params>, INT:ref, passed

is an array containing parameters describing any alternate keys for this file. This parameter is required if the file has alternate keys, otherwise it may be omitted or its first word must be zero. The format for this array is shown in the "<alternate key params> ARRAY" table which follows.

<partition params>, INT:ref, passed

is an array containing parameters that describe this file if the file is a multi-volume file. If the file is to span multiple volumes, then this parameter is required, otherwise this parameter may be omitted or its first word must be zero. The format for this array is shown in the "<partition params> ARRAY" table which follows.

condition code settings:

```
<  (CCL) indicates that the CREATE failed (call FILEINFO)
=  (CCE) indicates that the file was created successfully
>  (CCG) the device is not a disc
```

example

```
CALL CREATE(filename,5,0);
IF < THEN ...                    ! CREATE failed.
```

Considerations

- Disc Allocation at CREATE Time

  Execution of the CREATE procedure does not allocate any disc area; it only provides an entry in the volume's directory indicating that the file exists.

- Error Handling

  If the CREATE fails (i.e., condition code other than CCE returned), the reason for the failure can be determined by calling the FILEINFO procedure and passing -1 as the <file number> parameter.

- File Security

  The file is created with the default security associated with the process creator's access id. Security can be changed by opening the file and calling SETMODE or SETMODENOWAIT.

- Odd Unstructured Files

  When creating unstructured files, the value passed for <file type>.<l2> determines how all subsequent read, write, and position operations to the file will work.

  If <file type>.<l2> is passed as a 1, the values of <record specifier>, <read count>, and <write count> are all interpreted exactly. That is, a <write count> or <read count> of seven transfers exactly seven bytes.

  If <file type>.<l2> is passed as a 0, the values of <record specifier>, <read count>, and <write count> are all rounded up to an even number before the operation is performed. That is, a <write count> or <read count> of seven is rounded up to eight, and eight bytes are transferred.

- Key-Sequenced Parameter Array Format

The key-sequenced parameter array format is shown in Table 3-4.

Table 3-4. <key-sequenced params> ARRAY FORMAT

word:	
[0]	<key length>
[1]	<key offset>
[2]	<index block length>

where

<key length>, INT,

is the length, in bytes, of the record's primary key field

<key offset>, INT,

is the number of bytes from the beginning of the record where the primary key field starts.

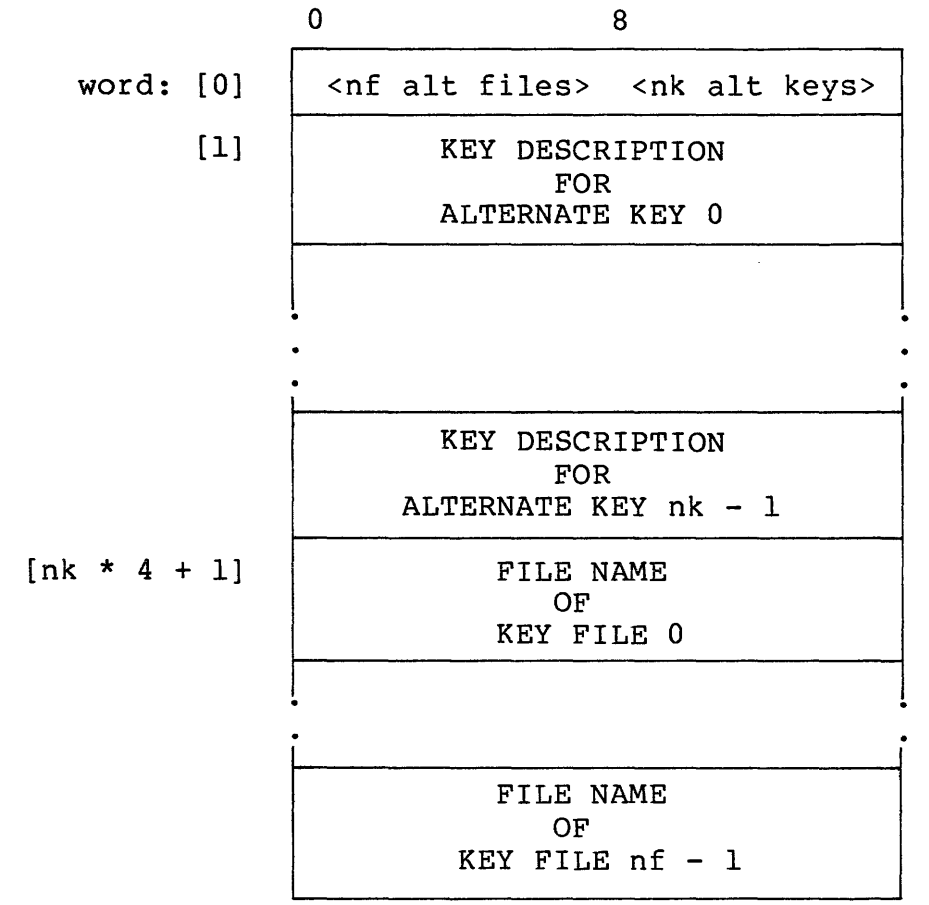
<index block length>, INT,

is the length in bytes of each index block in the file. <index block length> must be a multiple of 512 and may not be greater than 4096. If zero is specified, then the value of <data block length> is used as the <index block length>

- Alternate Key Parameter Array Format

The alternate key parameter array format is shown in Table 3-5.

Table 3-5. <alternate key params> ARRAY FORMAT



Key Description for key "k" consists of four words of the form:

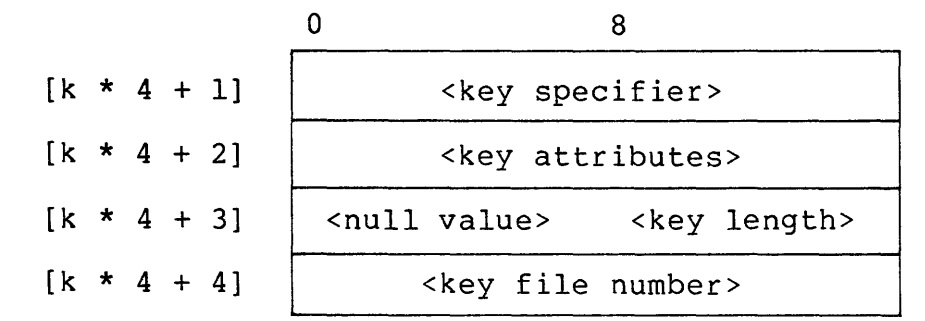


Table 3-5. <alternate key params> ARRAY FORMAT (cont'd)

where

<nf alt files>, one-byte value,

specifies the number of alternate key files for this primary file.

<nk alt keys>, one-byte value,

specifies the number of alternate key fields in this primary file.

<key specifier>, INT,

is a two-byte value that uniquely identifies this alternate key field. This value is passed to the KEYPOSITION procedure when referencing this key field.

<key attributes>, INT,

describes the key.

where

<key attributes>.<0>: 1 = null value is specified. See "<null value>" below.

<key attributes>.<1>: 1 = key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insert is rejected with a "duplicate record" error.

<key attributes>.<2>: 1 = no automatic updating of this key is to be performed by ENSCRIBE.

<key attributes>.<3> must be zero.

<key attributes>.<4:15> = <key offset>. This specifies the the number of bytes from the beginning of the record where this key field starts.

→

Table 3-5. <alternate key params> ARRAY FORMAT (cont'd)

<null value>, one-byte value,

is used to specify a "null value" if <key attributes>.<0> = 1.

During an insertion (i.e., WRITE), if a null value is specified for an alternate key field and the null value is encountered in all bytes of this key field, ENSCRIBE does not enter the reference to the record into the alternate key file. (If the file is read via this alternate key field, records containing a null value in this field will not be found.)

During a deletion (i.e., WRITEUPDATE, <write count> = 0), if a null value is specified and the null value is encountered in all bytes of this key field within <buffer>, ENSCRIBE deletes the record from the primary file but does not delete the reference to the record in the alternate file.

<key length>, one-byte value,

that specifies the length, in bytes, of this key field.

<key file number>, INT,

is the relative number in the <alternate key params> array of this key's alternate key file. The first alternate key file's <key file number> = 0.

The File Name for file "f" consists of 12 words and begins at

[nk * 4 + 1 + f * 12]

and is of the form

```
<file name[0:3]>  is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
<file name[4:7]>  is <subvol name><blank fill>
<file name[8:11]> is <disc file name><blank fill>
```

- Partition Parameter Array Format

The partition parameter array format is shown in Table 3-6.

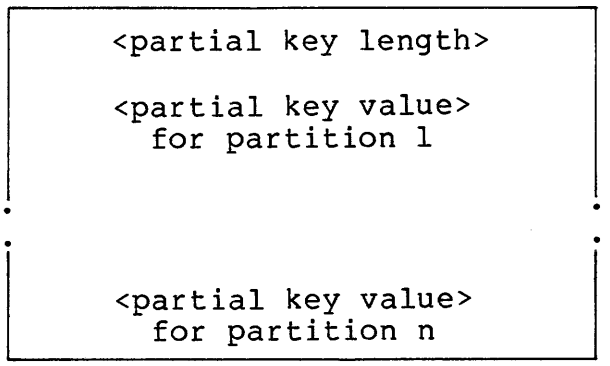
Table 3-6. <partition params> ARRAY FORMAT

	num words
<num extra partitions>	[1]
$<volume name> or \<sys num><volume name> for partition 1	[4]
$<volume name> or \<sys num><volume name> for partition 2	
. . .	
$<volume name> or \<sys num><volume name> for partition n	
<primary extent size> part 1	[1]
. . .	
<primary extent size> part n	
<secondary extent size> part 1	[1]
. . .	
<secondary extent size> part n	

→

Table 3-6. <partition params> ARRAY FORMAT (cont'd)

The following must be included in the <partition params> array for key-sequenced files and may be omitted for other file types:



[1]

where

<num extra partitions>, INT,

is the number of extra volumes (other than the one specified in the <file name> parameter) on which the file is to reside. The maximum value permitted is 15. Note that every other parameter in the partition array (except <partial key length>) must be specified <num extra partitions> times.

$<volume name> or
\<sys num><volume name>, eight bytes blank filled,

is the name of the disc volume (including "$" or "\") where the particular partition is to reside.

<primary extent size>, INT,

is the size of the primary extent for the particular partition.

<secondary extent size>, INT,

is the size of the secondary extents for the particular partition. Specifying zero results in the <primary extent size> value being used.

→

Table 3-6. <partition params> ARRAY FORMAT (cont'd)

The remaining parameters are required for key-sequenced files and may be omitted for all other file types.

<partial key length>, INT,

is the number of bytes of the primary key of a key-sequenced file that will be used to determine which partition of the file will contain a particular record. The minimum value for <partial key length> is one.

<partial key value>, INT,

for <partial key length> bytes, specifies the lowest key value that will be allowed for a particular partition.

Each <partial key value> in <partition params> must begin on a word boundary.

DEVICEINFO

The DEVICEINFO Procedure is used to obtain the device type and the physical record length for a file. The file may be open or closed.

The call to the DEVICEINFO Procedure is:

```
CALL DEVICEINFO (  <file name>
                 , <device type>
                 , <physical record length> )
```

where

<file name>, INT:ref, passed

is an array containing the name of the device whose characteristics are to be returned. Any form of <file name> is permitted. For disc files, only the first eight characters (i.e, the <volume name>) are significant, but the remaining eight characters must be blank.

<device type>, INT:ref:1, returned

is returned the device type of the associated file. See the "Device Type" table which follows.

<physical record length>, INT:ref:1, returned

is returned the physical record length associated with the file:

For non-disc devices, <physical record length> is the configured record length

For disc files, <physical record length> is the maximum possible transfer length. This is equal to the configured buffer size for the device; either 2,048 or 4,096 bytes

Note: The <logical record length> for an ENSCRIBE disc file is obtained via the FILERECINFO Procedure

For processes and the $RECEIVE file, 132 is returned in <physical record length> (this is the system convention for interprocess files)

→

condition code settings:

The condition code has no meaning following a call to DEVICEINFO.

example

```
CALL DEVICEINFO ( infile, devtype, reclength );
```

Device Types and Subtypes

The values returned for <device type> are listed in Table 3-7.

Table 3-7. DEVICE TYPES AND SUBTYPES

<device type>.<0>, 1 = removable disc volume

device type, <device type>.<4:9>,			device subtype, <device type>.<10:15>,
0	=	process	none
1	=	operator console	none
2	=	$RECEIVE	none
3	=	Disc	0 = 10 MB capacity 1 = 50 MB capacity 2 = 160 MB capacity 3 = 240 MB capacity 4 = 64 MB capacity (SMD) 5 = 64 MB capacity (MMD) 6 = 1 MB capacity 7 = 2 MB capacity
4	=	Magnetic Tape	1 = Nine track 2 = Seven track

→

Table 3-5. DEVICE TYPES AND SUBTYPES (cont.)

5	=	Line Printer	0	= Belt Printer
			1	= Drum
			2	= Current-Loop Belt
			3	= Current-Loop Matrix
			4	= Matrix Serial
6	=	Terminal (conversational or page mode)	0	= Character Mode
			1	= 6510
			2	= 6520
			32	= Hard Copy Console
7	=	Data Communication Line	0	= BISYNC, point-to-point, non-switched
			1	= BISYNC, point-to-point, switched
			2	= BISYNC, multipoint, tributary
			3	= BISYNC, multipoint, supervisor
			8	= ADM-2, multipoint, supervisor
			9	= TINET, multipoint, supervisor
			10	= Burroughs, multipoint, supervisor
			30	= Full-duplex, out line
			31	= Full-duplex, in line
			40	= Asynchronous line supervisor
			56	= Autocall unit
8	=	Punched Card Reader	none	
9	=	X.25 access method PTP protocol	none	
11	=	Bit synchronous ENVOY	40	= Synchronous Data Link Control (SDLC)
12	=	Tandem to IBM Link (TIL)	none	
26	=	Tandem Hyper Link	none	

EDITREAD

The EDITREAD procedure reads text lines from an edit file (filetype = 101). Before EDITREAD can be called, a call to EDITREADINIT must be completed successfully.

Text lines are transferred, in ascending order, from the text file to a buffer in the application program's data area. One line is transferred with each call to EDITREAD. EDITREAD also returns the sequence number associated with the text line and performs checks to ensure that the text file is valid.

The edit file can be opened with nowait i/o specified. However, a call to EDITREAD completes before returning to the application program; it is not completed with a call to AWAITIO.

The call to the EDITREAD procedure is:

```
<status> := EDITREAD (  <edit control block>
                      , <buffer>
                      , <buffer length>
                      , <sequence number>      )
```

where

<status>, INT, passed

is a value indicating the outcome of EDITREAD: Values for <status> are:

>= 0 indicates that the read was successful. This is the actual number of characters in the text line. However, only <buffer length> bytes are transferred into <buffer>.

< 0 indicates an unrecoverable error. Where:

- -1 = end-of-file encountered
- -2 = read error
- -3 = text file format error
- -4 = sequence error. The sequence number of the line just read is less than its predecessor

→

<edit control block>, INT:ref, passed

is the same array as specified in the parameter to EDITREADINIT.

<buffer>, STRING:ref, returned

is an array where the text line is to be transferred.

<buffer length>, INT:value, passed

is the length, in bytes, of the <buffer> array. This specifies the maximum number of characters in the text line that will be transferred into buffer.

<sequence number>, INT(32):ref, returned

is the sequence number multiplied by 1000, in doubleword integer form, of the text line just read.

example

```
count := EDITREAD(control^block, line, length, seq^num);
```

Extended EDITREADINIT and EDITREAD Example

The data declarations:

```
    .
  LITERAL buf^size = 512, ! EDITREAD's internal buffer size in bytes
          length   = 80;  ! length of the application's buffer (byte)

  INT   fnum,
        fcode,
        error,
        count,
       .control^block[0:39+buf^size/2]; ! global data declaration.

  STRING  .line[0:length/2-1];          ! application's buffer.

  INT(32) seq^num;
```

First the text file is opened and verified that it is an edit format file:

```
    .
    .
  CALL OPEN(fname,fnum,.);
  IF < THEN ...;
  CALL FILEINFO(fnum,,,,,,,,,fcode);
  IF fcode <> 101 THEN .. ! not edit format file.
    .
    .
```

Then EDITREADINIT is called to initialize the <edit control block> and specify EDITREAD's internal buffer size:

```
    .
    .
  IF (error := EDITREADINIT(control^block,fnum,buf^size)) THEN
    BEGIN ! unsuccessful
      .
      .
    END;
    .
    .
```

To read a text line, EDITREAD is called:

```
      .
  loop:
     IF (count := EDITREAD(control^block,line,length,seq^num)) < 0
     THEN
       BEGIN ! unsuccessful
         .
         .
       END
     ELSE
       IF count > length THEN ... ! line truncated.
       .
       .
     GOTO loop
```

If the read is successful, a count of the number of bytes in the text line is returned in "count", the text line is returned in the array "line", and the sequence number is returned in "seq^num".

Saving and Restoring a Location in an EDIT File

A specific location in an EDIT file can be saved during sequential reading and later be restored to reposition a process at the saved location. To save a location within an EDIT file, the second through fourth words of the edit control block are stored in a temporary buffer.

Later, repositioning to the saved location is done in two steps:

1) the current second through fourth words of the edit control block are replaced with the three words saved in the temporary buffer.

2) the first bit of the first word in the edit control block is set to a zero.

Calling EDITREAD after repositioning returns the next record after the saved position.

An example:

```
INT edit^cb[0:79],
    position[0:2];
        .
        .
        .
! EDITREADINIT and one or more EDITREADs are called
        .
        .
        .
position ':=' edit^cb[1] for 3;     ! save current position
        .
        .
        .
! more EDITREADs
        .
        .
        .
edit^cb[1] ':=' position for 3;     ! restore saved position
edit^cb.<0> := 0;

! next EDITREAD returns same record returned after position was
! saved
```

EDITREADINIT

The EDITREAD procedure is called to prepare a buffer in the application program's data area for subsequent calls to EDITREAD.

The application program designates an array to be used as an <edit control block>. The <edit control block> is used by the EDITREAD procedure for storing control information and for an internal buffer area.

The edit file can be opened with NOWAIT I/O. However, a subsequent call to EDITREADINIT completes before returning to the application; it is not completed with a call to AWAITIO.

The call to the EDITREADINIT procedure is:

```
<status> := EDITREADINIT (  <edit control block>
                          , <file number>
                          , <buffer length>       )
```

where

<status>, INT, returned

is a value indicating the outcome of EDITREADINIT. Values for <status> are:

```
   0 = successful (ok to read)
  -1 = end-of-file detected (empty file)
  -2 = input/output error
  -3 = format error (not EDIT file)
```

<edit control block>, INT:ref, passed

is an uninitialized array declared globally. Forty words of the <edit control block> are used for control information. The remainder is used as an internal buffer by EDITREAD. The length, in words, of the <edit control block> must be at least 40 + <buffer length> divided by 2.

<file number>, INT:value, passed

identifies the edit file to be read.

→

<buffer length>, INT:value, passed

is the size, in bytes, of the internal buffer area used by EDITREAD. This parameter determines the amount of data that EDITREAD reads from the text file on disc (not the amount of data transferred into the <buffer> specified as a parameter to EDITREAD). The size of the internal buffer area must be a power of two, from 64 to 2,048 bytes (i.e., 64, 128, 256, ..., 2,048).

example

```
INT .control^block[0:167];
n := EDITREADINIT(control^block, fnum, 256);
```

An extended example using both EDITREADINIT and EDITREAD is shown under the syntax description of EDITREAD.

FILEERROR

The FILEERROR procedure is used to determine if an i/o operation, that completed with an error, should be retried.

The call to the FILEERROR procedure is:

```
<status> := FILEERROR ( <file number> )

where

   <status>, INT,                                          returned

     has two possible values:

       0 = operation shouldn't be retried (i.e., error is fatal)
       1 = operation should be retried


   <file number>, INT:value,                                passed

     identifies the file having the error.


example

   IF FILEERROR(fnum) THEN .. ! retry
```

The FILEERROR procedure is called after a CCL return from a file management procedure. The FILEERROR procedure determines if an operation should be retried as follows:

- FILEERROR obtains the file management error number and file name through a call to the FILEINFO procedure, or
- If the error is caused by a disc pack not up to speed

  FILEERROR delays the calling process for one second then returns a one indicating a retry should be performed.

- If the error is caused by a device not ready, an appropriate message is printed on the home terminal. This is followed by a read from the terminal. If, at this time, "STOP" is entered (signalling that the condition cannot be corrected), FILEERROR returns a zero indicating that the operation should not be retried. If any other data is entered (typically, carriage return), it signals that the condition has been corrected, and FILEERROR returns a one indicating that the operation should be retried.

- If the error is caused by an ownership error (<error> = 200) or path down error (<error> = 201) and the alternate path is operable, FILEERROR returns a one indicating the the operation should be retried. If the alternate path is inoperable, a zero is returned.

- Any other error results in the file name being printed on the home terminal followed by the file management error number. A zero is returned indicating that the operation should not be retried.

Two examples follow. First example:

```
error := 1;
WHILE error DO
  BEGIN
    CALL WRITE(fnum,buffer,count);
    IF < THEN
      BEGIN
        IF NOT FILEERROR(fnum) THEN CALL ABEND;
      END
    ELSE error := 0;
  END;
```

It may be desirable to check for certain errors before calling FILEERROR. Therefore, the program itself should first call FILEINFO:

```
LITERAL nofile = 11;     ! data declaration.

not^open := not^created := 1;
WHILE not^open DO ! open the file.
  BEGIN
    CALL OPEN(fname, fnum, 0);
    IF < THEN
      BEGIN
        CALL FILEINFO(fnum,error);
        IF error = nofile THEN ! file does not exist. create it
          WHILE not^created DO
            BEGIN
              CALL CREATE(fname, ..);
              IF < THEN ! creation failure
                BEGIN
                  IF NOT FILEERROR(-1) THEN CALL ABEND;
                END
              ELSE not^created := 0;
            END
        ELSE
          BEGIN
            IF NOT FILEERROR (fnum) THEN CALL ABEND;
          END;
      END
    ELSE not^open := 0;
  END; ! open the file.
```

FILEINFO

The FILEINFO procedure is used to obtain error and characteristic information about an open file.

The call to the FILEINFO procedure is:

```
CALL FILEINFO (     <file number>
                , [ <error>                    ]
                , [ <file name>                ]
                , [ <logical device number>    ]
                , [ <device type>              ]
                , [ <extent size>              ]
                , [ <end-of-file location>     ]
                , [ <next-record pointer>      ]
                , [ <last mod time>            ]
                , [ <file code>                ]
                , [ <secondary extent size>    ]
                , [ <current-record pointer>   ]
                , [ <open flags>               ] )
```

where

<file number>, INT:value, passed

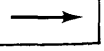
identifies the file whose characteristics are to be returned.

<error>, INT:ref:1, returned

if present, is returned the error number associated with the last operation on the file (see "Errors and Error Recovery").

<file name>, INT:ref:12, returned

if present, is returned the file name of this file. See "File Names" for the file name format.



<logical device number>, INT:ref:16, returned

if present, is returned the logical device number of the device where this file resides (in binary).

For partitioned files, an array of <logical device numbers> is returned; one entry for each of 16 possible partitions:

[0] = <logical device number> of partition 0
[1] = <logical device number> of parititon 1
.
.
.
[15] = <logical device number> of partition 15

If -1 is returned for a partition, then the partition is not open.

<device type>, INT:ref:1, returned

if present, is returned the device type of the device associated with this file. See "DEVICEINFO Procedure", table of "Device Types and Subtypes"

<extent size>, INT:ref:1, returned

if present, is returned the primary extent size in 2048 byte units.

<end-of-file pointer>, INT(32):ref:1, returned

if present, is returned the relative byte address of the end-of-file location.

<next-record pointer>, INT(32):ref:1, returned

if present, is returned the next-record pointer setting.

For relative files, this is a <record number>; for entry-sequenced files, this is a <record address>; for unstructured files, this is an <rba>; for key-sequenced files, this parameter is ignored (i.e., whatever is passed is returned unchanged).

<last mod time>, INT:ref:3, returned

if present, is returned a three-word timestamp indicating the last time that the file was modified. <last mod time> is of the same form as the <interval clock> returned by TIMESTAMP and can be converted into a date by CONTIME.

<file code>, INT:ref:1, returned

if present, is returned the application defined file code that was assigned when the file was created. File codes 100-999 are reserved for use by Tandem Computers, Inc.

<secondary extent size>, INT:ref:1, returned

if present, is returned the size of the secondary file extents (extents 1-15) in 2048 byte units

<current-record pointer>, INT(32):ref:1, returned

if present, is returned the setting of the current-record pointer. This may be an even or odd value.

For relative files, this is a <record number>; for entry-sequenced files, this is a <record address>; for unstructured files, this is an <rba>; for key-sequenced files, this parameter is ignored (i.e., whatever is passed is returned unchanged).

→

<open flags>, INT:ref:1, returned

if present, is returned the access granted when the file was opened. Where:

<open flags>.<12:15> is the maximum number of concurrent no-wait i/o operations that can be in progress on this file at any given time. <open flags>.<12:15> = 0 implies "wait i/o".

<open flags>.<9:11> is the exclusion mode:

0 = shared access
1 = exclusive access
3 = protected access

<open flags>.<8> 1 = for process files, the OPEN message is sent no-wait and must be completed by a call to AWAITIO.

<open flags>.<6> 1 = resident buffers have been provided by the application process for calls to file system i/o routines. Resident buffering only applies to the Tandem NonStop System; a zero is always returned in this bit for the NonStop II System (see "OPEN Procedure").

<open flags>.<3:5> is the access mode:

0 = read/write access
1 = read-only access
2 = write-only access

<open flags>.<2> 1 = unstructured access regardless of the actual file structure (see "OPEN Procedure").

condition code settings:

< (CCL) indicates that an error occurred, the error number is returned in <error>
= (CCE) indicates that FILEINFO executed successfully
> (CCG) is not returned by FILEINFO

example

```
CALL FILEINFO ( infile, err^num );
```

Considerations

- Error Handling

  The error number of a preceding AWAITIO on any file or waited OPEN that failed can be obtained by passing a -1 in the <file number> parameter. The error number is returned in <error>.

- Calling FILEINFO Before Opening any Files

  <error> = 32 is returned in <error> (if <error> is a parameter present in the call) if a process has never opened any files and -1 is specified in the <file number> parameter.

- Commas (A word of Caution)

  All parameters to FILEINFO, except <file number>, are optional. Placeholder commas must be included to indicate missing parameters; commas can be omitted for rightmost missing parameters.

```
.
CALL FILEINFO ( devicenum, error,,, devicetype,, eof );
.
```

- Error 16: File Not Open

  Calling FILEINFO subsequent to a close, returns <error> = 16, file not open.

- Error Recovery for a Failed CREATE or PURGE

  The error number of a preceding CREATE or PURGE that failed can be obtained by passing a -1 in the <file number> parameter. The error number is returned in <error>.

FILERECINFO

The FILERECINFO procedure is used to obtain the characteristics of an open Enscribe disc file.

The call to the FILERECINFO procedure is:

```
CALL FILERECINFO (    <file number>
                  , [ <current key specifier>      ]
                  , [ <current key value>          ]
                  , [ <current key length>         ]
                  , [ <current primary key value>  ]
                  , [ <current primary key length> ]
                  , [ <partition in error>         ]
                  , [ <specifier of key in error>  ]
                  , [ <file type>                  ]
                  , [ <logical record length>      ]
                  , [ <block length>               ]
                  , [ <key-sequenced params>       ]
                  , [ <alternate key params>       ]
                  , [ <partition params>           ] )
```

where

<file number>, INT:value, passed

identifies the file whose characteristics are to be returned.

<current key specifier>, INT:ref:1, returned

if present, is returned the key specifier of the current key field.

<current key value>, STRING:ref:*, returned

if present, is returned the value of the current key for <current key length> bytes.

<current key length>, INT:ref:1, returned

if present, is returned the is length, in bytes, of the current key.

→

<current primary key value>, STRING:ref:*, returned

if present, is returned the value of the current primary key for <current primary key length> bytes.

<current primary key length>, INT:ref:1, returned

if present, is returned the is length, in bytes, of the current primary key

<partition in error>, INT:ref:1, returned

if present, is returned a number from 0 through 15 indicating the partition associated with the latest error occuring with this file.

<specifier of key in error>, INT:ref:1, returned

if present, is returned the key tag associated with the latest error occurring with this file.

<file type>, INT:ref:1, returned

if present, is returned indicating the type of file being accessed:

where

<file type>.<13:15> specifies the file structure:

0 = unstructured
1 = relative
2 = entry-sequenced
3 = key-sequenced

<file type>.<12> 1 = 'ODDUNSTR' is specified for unstructured files.

<file type>.<12> 1 = data compression is specified for key-sequenced files.

<file type>.<11> 1 = index compression is specified for key-sequenced files.

→

<file type>.<2> 1 = for systems with the Transaction Monitoring Facility (TMF), indicates file is audited.

<logical record length>, INT:ref:1, returned

if present, the maximum size of the logical record in bytes is returned.

<block length>, INT:ref:1, returned

if present, is returned the length, in bytes, of a block of records for the file

<key-sequenced params>, INT:ref:*, returned

if present, is an array where the parameters unique to an key-sequenced file are returned. See the description under "CREATE Procedure" in this section.

<alternate key params>, INT:ref:*, returned

if present, is an array where the parameters describing the file's alternate keys are returned. See the description under "CREATE Procedure" in this section.

<partition params>, INT:ref:*, returned

if present, is an array where the parameters describing a multi-volume file are returned. See the description under "CREATE Procedure" in this section.

condition code settings:

< (CCL) indicates that an error occurred
= (CCE) indicates that FILERECINFO executed successfully
> (CCG) indicates that the file is not an Enscribe disc file

example

```
CALL FILERECINFO ( infile,,,,,,,, ftype );
```

→

FNAMECOLLAPSE

The FNAMECOLLAPSE procedure converts a file name from its internal form to its external form. The system number of a network file name is converted to the corresponding system name.

The call to the FNAMECOLLAPSE procedure is:

```
{ length := }  FNAMECOLLAPSE (  <internal name>
                              , <external name> )
```

where

<length>, INT, returned

is returned the number of bytes in <external name>.

<internal name>, INT:ref:12, passed

is the name to be converted. If this is in local form, it is converted to external local form; if it is in network form, it is converted to external network form. Network file names are discussed in the "File Names" section of this manual.

<external name>, STRING:ref:26 or 34 returned

contains, on return, the external form of <internal name>. If <internal name> is a local file name, <external name> contains 26 bytes; if a network name is converted, <external name> contains 34 bytes.

example

```
length := FNAMECOLLAPSE( internal, external );
```

Examplee of File Name Conversion

```
local:   $SYSTEM SUBVOL   MYFILE
         is converted to "$SYSTEM.SUBVOL.MYFILE"

network: \<sysnum>SYSTEMSUBVOL   MYFILE
         is converted to "\<system name>.$SYSTEM.SUBVOL.MYFILE
```

Considerations

- Passing Invalid File Names

  It is the responsibility of the program calling FNAMECOLLAPSE to pass a valid file name in <internal name>. Invalid file names cause unpredictable results.

- Passing a Bad <sysnum> Value

  If <internal name> is in network form and the system number in the second byte does not correspond to any system in the network, FNAMECOLLAPSE supplies "???????" as the system name.

FNAMECOMPARE

The FNAMECOMPARE procedure compares two file names within a local or network environment to determine whether these file names refer to the same file or device. For example, one name may be a logical system name or a device number while the other reference is a symbolic name. The file names compared must be in the standard twelve-word internal format that is returned by FNAMEEXPAND.

The call to the FNAMECOMPARE procedure is:

```
{ status := } FNAMECOMPARE (  <file name 1>
{ CALL     }                , <file name 2> )
```

where

<status>, INT, returned

is a value indicating the outcome of the comparison. Values for <status> are:

-1	=	(CCL) the file names do not refer to the same file
0	=	(CCE) the file names refer to the same file
+1	=	(CCG) the file names refer to the same <volume name>, <device name>, or <process name> on the same system, however, words [4:11] are not the same: <file name 1> [4] <> <file name 2> [4] FOR 8.

A value less than negative one is the negative of a file management error code. This indicates that the comparison is not attempted due to this error condition.

That value returned from the program function determines the condition code setting.

<file name 1>, INT:ref:12, passed

the first comparable file name. Each <file name> array may contain either a local file name or a network file name. Definitions of file names are found in the GUARDIAN OPERATING SYSTEM PROGRAMMING MANUAL, File Names section.

<file name 2>, INT:ref:12 passed

the second comparable file name.

Considerations

- File Name Arrays

  The arrays containing the file names for comparison are not modified.

- Alphabetic Character Handling

  Alphabetic characters within qualified process names are not upshifted before comparison.

- Passing Logical Device Numbers for File Names

  If a logical device number format such as $0076, is used for one file name, but not for the second file name, then the device table of the referenced system is consulted to determine whether the names are equivalent. This is the only case where the device table is used. All other comparisons involve only the examination of the two file names supplied.

- Common Errors Returned From FNAMECOMPARE

  Some of the most common negative file management error codes returned are:

  -13 = an illegal file name specification for either file name is made.

  -14 = the device does not exist. (See note.)

  -18 = no such system is defined in this network. (See note.)

  -22 = a parameter or buffer is out of bounds.

  -250 = all paths to the system are down. (See note.)

  Note: These negative file management error codes indicate that one file name is passed in logical device number format while the second is not and the device is connected to a remote network node.

Extended Example of Using FNAMECOMPARE

In the following example, the notation <x> refers to a number, not to an ASCII character; <%52> :==: "*".

Assume the following declarations:

```
INT  .fname1[ 0:11 ],
     .fname2[ 0:11 ],
      status;
```

Then in a network node with a system number of <6>, execution of

```
fname1 ':=' [ "$term1", 9 * [ "  "] ];
fname2 ':=' [ %56006, "TERM1 ", 8 * [ "  "] ];  ! "\", <6>, "TERM1"
status := FNAMECOMPARE ( fname1, fname2 );
```

returns a status of 0, and the condition code (CCE).

In a non-network system, execution of the above example returns a status of negative one, and the condition code (CCL).

Whether a system is a network node or not, execution of

```
fname1 ':=' [ "$SERVR  #START  UPDATING" ];
fname2 ':=' [ "$SERVR  #FINISH UPDATING" ];
status := FNAMECOMPARE ( fname1, fname2 );
```

returns a status of plus one, and the condition code (CCG).

In any system, execution of

```
fname1 ':=' [ "$0013 ", 9 * [ "  "] ];
fname2 ':=' [ "$DATAX", 9 * [ "  "] ];
status := FNAMECOMPARE ( fname1, fname2 );
```

returns a status of zero and condition code (CCE), if the device name $DATAX is defined as logical device number 13 at SYSGEN time, otherwise a status of negative one and the condition code (CCL) is returned.

FNAMECOMPARE can also verify the specified file names, as shown in the following example:

```
! assume all variables and procedures have been
! properly defined and initialized elsewhere
!
! also assume  LITERAL  legal = 0;

IF FNAMEEXPAND ( external^name, internal^name, default^names ) THEN
   BEGIN
   ! something reasonable was entered.
   IF FNAMECOMPARE ( internal^name, internal^name ) = legal THEN
      ! it may not exist, but looks okay.
      BEGIN
        .
        ! normal processing.
        .
      END
   ELSE
      ! the format is not legal.
      BEGIN
        .
        ! error processing.
        .
      END;
   END;
```

FNAMEEXPAND

The FNAMEEXPAND procedure is used to expand a partial file name from the compacted external form to the standard twelve-word internal form usable by file management procedures.

The call to the FNAMEEXPAND procedure is:

```
{ <length> := } FNAMEEXPAND (   <external file name>
                               , <internal file name>
                               , <default names>      )
```

where

<length>, INT, returned

is the length in bytes of the file name in <external file name>. If an invalid file name is specified, zero is returned.

<external file name>, STRING:ref, passed

is the file name to be expanded. The file name must be in the form

[\<system name>]<file name>

where <file name> is in one of these forms:

[$<volume name>.][<subvol name>.]<disc file name><delim>
$<device name><delim>
$<logical device number><delim>

<delim>

is a delimiter character. <delim> can be any character that is not valid as part of an <external file name> such as <blank> or <null>.

<internal file name>, INT:ref, returned

is an array of twelve words where FNAMEEXPAND returns the expanded file name. This cannot be the same array as <external file name>.

→

```
   <default names>, INT:ref,                                  passed

     is an array of eight words containing the default volume and
     subvol names to be used in file name expansion.  <default
     names> is of the form:

       <default names[0:3]>  = default <volume name> (blank
                               filled on right)
       <default names[4:7]>  = default <subvol name> (blank
                               filled on right)

     <default names[0:7]> corresponds directly to <word[1:8]> of
     the Command Interpreter parameter message.  See the Guardian
     Programming Manual for the parameter message format.


example

   length := FNAMEEXPAND(inname,outname,pmsg[1]);
```

Examples of File Name Expansion by FNAMEEXPAND

```
   <disc file name> is returned as

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

   <subvol name>.<disc file name> is returned as

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

   $<volume name>.<disc file name> is returned as

      <file name[0:3]>  = $<volume name><blank fill>
      <file name[4:7]>  = <default subvol name><blank fill>
      <file name[8:11]> = <disc file name><blank fill>
```

$<volume name>.<subvol name>.<disc file name> is returned as

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

$<device name> is returned as

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

$<logical device number> is returned as

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

any other file name is invalid

Extended Example Using FNAMEEXPAND

Assuming the following declarations:

```
STRING .ext^names[0:24] := "  filea   $system.fileb  ",
       .p; ! string pointer.

INT .infile[0:11],
    .outfile[0:11],
    .defaults[0:7] := "$voll   ",
                      "svoll   ";
```

FNAMEEXPAND is used to expand the external file names into a usable internal form:

```
    .
SCAN ext^name WHILE " " -> @p; ! skip leading blanks.
@p := FNAMEEXPAND(p, infile, defaults) + @p;
```

on the completion of FNAMEEXPAND, <infile> contains

```
"$voll   svoll   filea   "
```

which is suitable for passing to the file management CREATE, OPEN, RENAME, and PURGE procedures as well as the process control NEWPROCESS procedure.

"p" is incremented by the number of characters in the external file name.

```
    .
SCAN p WHILE " " -> @p; ! skip intermediate blanks.
CALL FNAMEEXPAND(p, outfile, defaults);
    .
```

on the completion, "outfile" contains

```
"$system svoll   fileb   ".
```

Expanding Network File Names

FNAMEEXPAND converts local file names to local names, and network file names to network names. Network file names are described under "File Names".

When network file names are involved, in addition to expanding the local part of the name using the defaults, FNAMEEXPAND converts the system name to the appropriate system number.

Example:

Suppose that system \NEWYORK is assigned system number 4. Then the external file name "\NEWYORK.$DATA.SUB.MYFILE" is converted by FNAMEEXPAND to

```
\<%4>DATA  SUB     MYFILE
```

where "<%4>" denotes octal 4 in the second byte.

The use of FNAMEEXPAND in programming network applications is discussed fully in the EXPAND User's Manual.

GETDEVNAME

The GETDEVNAME procedure is used to obtain the name associated with a logical device number. GETDEVNAME returns, from the Logical Device Table (LDT), the name of a designated logical device if such a device exists or the name of the next higher (numerically) logical device if the designated logical device does not exist. A status word is returned from GETDEVNAME that indicates whether or not the designated device exists or if higher entry exists in the LDT. By repeatedly calling GETDEVNAME and supplying successively higher logical device numbers, the names of all system device can be obtained.

The call to the GETDEVNAME procedure is:

```
<status> := GETDEVNAME (    <logical device no>
                        ,   <device name>
                        , [ <system number>     ] )
```

where

<status>, INT returned

indicates the outcome of the call. Where

- 0 = successful, the name of the designated logical device is returned in <device name>
- 1 = the designated logical device does not exist. The logical device number of the next higher device is returned in <logical device no>; the name of that device is returned in <device name>
- 2 = "end-of-LDT", there is no logical device equal-to or or greater than <logical device no>
- 3 = unable to get name for demaountable disc
- 4 = the system specified could not be accessed
- 99 = parameter error

<logical device no>, INT:ref:1, passed, returned

on the call, is passed the logical device number, in binary, of the designated logical device whose name is to be returned.

On the return, <logical device no> is is returned the logical device number, in binary, of the device whose name is actually returned. If "end-of-LDT" is encountered, <logical device no> is unchanged.

→

<device name>, INT:ref:4, returned

is returned the <device name> or <volume name> of the designated device if it exists or the next higher logical device if the designated device does not exist. If "end-of-LDT" is encountered, <device name> is unchanged.

<system number>, INT, passed

if present, specifies the system (in a network) whose Logical Device Table is to be searched for <logical device no>.

If absent, the local system is assumed.

condition code settings:

The condition code setting has no meaning following a call to GETDEVNAME.

example

```
! get the names of all logical devices.
ldev := 0;
WHILE NOT GETDEVNAME ( ldev , devname ) DO
  BEGIN
    CALL print ( ldev , devname );
    ldev := ldev + 1;
  END;
```

Considerations

- Specifying Remote Logical Devices

  If the device specified by <logical device no.> is remote, its <device name> is returned in network form; otherwise, the <device name is returned in local form.

- Limitations When <system number> is Passed

  If the <system number> parameter is supplied, devices whose names contain seven characters are not accessible using this procedure.

KEYPOSITION

The KEYPOSITION procedure is used to position by primary key within key-sequenced files, and by alternate key within key-sequenced, relative and entry-sequenced files.

KEYPOSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The calling application process is not suspended because of a call to KEYPOSITION.

A call to the KEYPOSITION procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the KEYPOSITION procedure is:

```
CALL KEYPOSITION (    <file number>
                  ,   <key value>
                  , [ <key specifier>    ]
                  , [ <length word>      ]
                  , [ <positioning mode> ] )
```

where

<file number>, INT:value, passed

identifies the file to be positioned.

<key value>, STRING:ref, passed

is the value which defines the current position in the file. The current position is found by a search of the access path specified by <key specifier>. The first record having an access path key field value that matches <key value>, as defined by <positioning mode> and <compare length>, becomes the current position.

→

<key specifier>, INT:value, passed

designates the key field to be used as the access path for the file:

<key specifier> = 0 or omitted, means use the file's primary key as the access path.

<key specifier> = predefined key specifier for an alternate key field, means use that field as the access path.

<length word>, INT:value, passed

contains two values, the <compare length> in the left byte <length word>.<0:7>, and the <key length> in the right byte <length word>.<8:15>.

<0:7>, the <compare length>, is the number of bytes of <key value> compared with the specified key field in the file. If omitted or zero, <compare length> is assumed to equal the minimum of the <key length> value and the key length defined for the file when it was created. If a shorter key length than that defined for the file is specified, the results are determined by the <positioning mode>.

<8:15>, the <key length>, is the number of bytes of <key value> searched for in the file to find the initial position. If omitted, the <key length> is assumed to equal the key length defined at file creation.

```
    <positioning mode>, INT:value,                          passed

      <positioning mode>.<0>   if 1 and a record with exactly the
      key specified is found, it is skipped.

      <positioning mode>.<14:15> indicate the type of key search
      (and, therefore, a subset of records),

      where

        0 = approximate  - positioning occurs to the first record
            whose key field, as designated by the <key specifier>,
            contains a value equal to or greater than <key value>
            for <compare length> bytes

        1 = generic - positioning occurs to the first record whose
            key field, as designated by the <key specifier>,
            contains a value equal to <key value> for <compare
            length> bytes

        2 = exact - positioning occurs to the first record whose
            key field, as designated by the <key specifier>,
            contains a value of exactly <compare length> bytes and
            is equal to <key value>

      If <positioning mode> is omitted, approximate is used.


  condition code settings:

     <  (CCL) indicates that an error occurred (call FILEINFO)
     =  (CCE) indicates that the KEYPOSITION was successful
     >  (CCG) no operation, not an Enscribe disc file


  example

     key ':=' "DOE,JOHN";
     CALL KEYPOSITION ( infile, key,, 8 );
     IF < THEN ....                    ! error occurred
```

Considerations

- Positioning on Duplicate or Nonexistent Records

  No searching of indices is done by KEYPOSITION. Therefore a nonexistent or duplicate record is not reported until a subsequent READ, READUPDATE, WRITEUPDATE, LOCKREC, READLOCK, READUPDATELOCK, or WRITEUPDATEUNLOCK is performed.

- KEYPOSITION and Disc Seeks

  KEYPOSITION does not cause the disc heads to be repositioned; the heads are repositioned when a subsequent i/o call (READ, READUPDATE, WRITE, etc.) transfers data.

- Positioning Exact

  If an exact KEYPOSITION is performed and a <compare length> is specified that is less than that specified when the file was created, <compare length> must match the variable key length specified when the record was entered into the file. Otherwise a subsequent call to READ, READUPDATE, WRITEUPDATE, etc., is rejected.

- Current State Indicators After a KEYPOSITION

Current state indicators following a successful KEYPOSITION:

current position	is that of the record indicated by the <key value>, <key specifier>, <positioning mode>, and <compare length>; or the subsequent record if <positioning mode>.<0> is set to 1.
positioning mode	is <positioning mode> if the parameter is supplied, otherwise approximate.
compare length	is <compare length> if the <length word> parameter is supplied, otherwise the defined length of the specified key field.

The compare length for generic searches is determined as follows:

```
IF <length word>.<0:7> <> 0
THEN <length word>.<0:7>
ELSE
   IF <length word>.<8:15> > length of <key specifier>
   THEN length of <key specifier>
ELSE <length word>.<8:15>
```

current primary key value	is <key value> if <key specifier> is primary, otherwise unchanged.

- Saving Current Position for Later Access

  To return to a position in a key-sequenced file, when processing by alternate key, save the concatenated alternate key and primary key values in a temporary buffer. For example:

```
<temporary buffer> ':=' record.altkey field for $len
                        (record.altkey field) and
                        record.primary key for $len
                        (record.primary key)
```

  Repositioning to the same record is done with:

```
KEYPOSTION ( <filenum>,
             <temporary buffer>,
             <key specifier>,
             <compare length for generic searches '<<' 8 +
             length of alternate key + length of primary key>,
             <positioning mode> ).
```

  Repositioning to the next record is done with:

```
KEYPOSTION ( <filenum>,
             <temporary buffer>,
             <key specifier>,
             <compare length for generic searches '<<' 8 +
             length of alternate key + length of primary key>,
             <%100000 + positioning mode> ).
```

  The <key specifier> specifies the alternate key.

LOCKFILE

The LOCKFILE procedure is used to temporarily exclude other accesses to a file.

If the file is currently unlocked or is locked by the caller when LOCKFILE is called, the file becomes locked and the caller continues executing.

Two "locking" modes are available:

- With the default mode, if the file is already locked when the call to LOCKFILE is made, the process requesting the lock is suspended and queued in a "locking" queue behind any other processes also requesting to lock or read the file. When the file becomes unlocked, the process at the head of the locking queue is granted access to the file. If the process at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the process at the head of the locking queue is requesting a read, the read operation continues to completion.

- With the alternate mode, if the file is already locked when the call to LOCKFILE is made, the lock request is rejected and the call to LOCKFILE completes immediately with a "file is locked" error indication (<error> = 73). The alternate locking mode is established by calling SETMODE and specifying function 4, set lock mode.

If the LOCKFILE procedure is being used to initiate an operation on a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure. Note that process suspension due to a queued lock occurs when AWAITIO is called and the alternate locking mode error "file is locked" is returned by AWAITIO (if the file was already locked).

The call to the LOCKFILE procedure is:

```
CALL LOCKFILE (    <file number>
               , [ <tag>         ] )

where

   <file number>, INT:value,                              passed

     identifies the file to be locked.
```

→

```
   <tag>, INT(32):value,                                    passed

     for no-wait i/o only, if present, is stored by the system,
     then passed back to the application process by the AWAITIO
     procedure when the lock operation completes.


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the LOCKFILE was successful
   >  (CCG) file is not a disc file


example

   CALL LOCKFILE ( file^num );
   IF < THEN .....;            ! error
```

Considerations

- Locks and Multiple Opens by the Same Process

  Locks are granted on an open file (i.e., <file number>) basis. Therefore, if a process has multiple opens of the same file, a lock of one <file number> excludes accesses to the file through other <file numbers>.

- Attempting to Write to A Locked File

  If a call to WRITE, or WRITEUPDATE is made and the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file is locked" error indication (<error> = 73).

- Attemting to Read a Locked File -- Default Locking Mode

  If the default locking mode is in effect when a call to READ or READUPDATE is made and the file is locked but not locked through the <file number> supplied in the call, the caller of READ or READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file.

  Note that a deadlock condition occurs if a call to READ or READUPDATE is made by the process having a file locked but the file is not locked via the <file number> supplied to READ or READUPDATE.

- Attempting to Read a Locked File -- Alternate Locking Mode

  If the alternate locking mode is in effect when READ or READUPDATE is called and the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file is locked" error indication (<error> = 73).

- Attempting to Control a Locked File

  If a call to CONTROL is made and the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file is locked" error indication (<error> = 73).

- Specifiying the Locking Mode

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

- Locks Are Not Nested

  Locks are not nested. For example:

```
.
CALL LOCKFILE ( file^a );
  .
  "file^a" becomes locked.
  .
CALL LOCKFILE ( file^a );
  .
  is a "null" operation because the file is already locked.  A
  condition code of CCE is returned.
  .
CALL UNLOCKFILE ( file^a );
  .
   "file^a" becomes unlocked.
  .
CALL UNLOCKFILE ( file^a );
  .
  is a "null" operation because file is already unlocked.  A
  condition code of CCE is returned.
```

LOCKREC

The LOCKREC procedure is used to temporarily exclude other accesses to the record at the current position. For key-sequenced, relative, and entry-sequenced files, the current position is the record with a key value that matches the current key value exactly. For unstructured files, the current position is the record identified by the current-record pointer.

If the record is either unlocked or is currently locked by the caller when LOCKREC is called, the record becomes locked and the caller continues executing.

Two "locking" modes are available:

- With the default mode, if the record is already locked when the call to LOCKREC is made, the process requesting the lock is suspended and queued in a "locking" queue behind any other processes also requesting to lock or read the record. When the record becomes unlocked, the process at the head of the locking queue is granted access to the record. If the process at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the process at the head of the locking queue is requesting a read, the read operation continues to completion.

- With the alternate mode, if the record is already locked when the call to LOCKREC is made, the lock request is rejected and the call to LOCKREC completes immediately with a "record is locked" error indication (<error> = 73). The alternate locking mode is specified via an option to the SETMODE procedure.

Note: A call to LOCKFILE is equivalent to locking all records in a file. Therefore, a file lock is queued behind any pending record locks. Conversely, a record lock is queued behind any pending file locks.

If the LOCKREC procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure. Additionally, the process suspension due to a queued lock occurs when AWAITIO is called.

The syntax for the LOCKREC procedure is shown on the following page.

The call to the LOCKREC procedure is:

```
CALL LOCKREC (    <file number>
              , [ <tag>         ] )

where

   <file number>, INT:value,                                  passed

     identifies the file containing the record to be locked.


   <tag>, INT(32):value,                                      passed

     for no-wait i/o only, if present, is stored by the system,
     then passed back to the application process by the AWAITIO
     procedure when the operation completes.


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the LOCKREC was successful
   >  (CCG) file is not a disc file


example

   CALL LOCKREC ( file^num, lock^tag );
   IF < THEN .....;          ! error
```

General Considerations

- Attempting to Write a Locked Record

  If a call to WRITE or WRITEUPDATE is made for a record and that record is locked but not through the <file number> supplied in the call, the call is rejected with a "record is locked" error indication (<error> = 73).

- Attempting to Read a Locked Record -- Default Locking Mode

  If the default locking mode is in effect when a call to READ or READUPDATE is made for a record and that record is locked but not locked through the <file number> supplied in the call, the caller to READ or READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to lock or read the record.

  Note that a deadlock condition occurs if a call to READ or READUPDATE is made by the process having a record locked but the record is not locked via the <file number> supplied to READ or READUPDATE.

- Attempting to Read a Locked Record -- Alternate Locking Mode

  If the alternate locking mode is in effect when READ or READUPDATE is called for a record and that record is locked but not through the <file number> supplied in the call, the call is rejected with a "record is locked" error indication (<error> = 73).

- Attempting to Control a File Containing a Locked Record

  If a call to CONTROL is made for a file containing a record that is not locked through the <file number> supplied in the call, the call is rejected with a "record is locked" error indication (<error> = 73).

- Selecting the Locking Mode with SETMODE

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

- Locks Can Not Be Nested

  Locks are not nested. As an example:

```
CALL LOCKREC ( file^a, .. );
     .  (locks the current-record in "file^a")
     .
CALL LOCKREC ( file^a, .. );
     .  (has no effect since current record already locked)
     .
CALL UNLOCKREC ( file^a, .. );
     .  (unlocks the current-record in "file^a")
     .
CALL UNLOCKREC ( file^a, .. );
     .  (has no effect since current record is not locked)
     .
```

Considerations for Structured Files

- Calling LOCKREC after Positioning on a Nonunique Key

  If the call to LOCKREC immediately follows a call to KEYPOSITION where a non-unique alternate key is specified, the LOCKREC fails. A subsequent call to FILEINFO returns error 46 (invalid key). However, if an intermediate call to READ is performed, the call to LOCKREC is permitted because a unique record is identified.

- Current State Indicators After LOCKREC

  Current state indicators following a successful LOCKREC:

  unchanged.

Considerations for Unstructured Files

- Locking Records in an Unstructured File

  Record positions in an unstructured file, represented by a relative byte address (rba), can be locked with LOCKREC. To lock a record position in an unstructured file, first position to the record by calling POSITION with the desired rba, and then call LOCKREC. This locks the rba; any other process attempting to access the file with exactly the same rba will encounter a "record is locked condition". Depending on the process's locking mode, the process's call will either fail with error 73, record is locked, or be placed on the locking queue.

- Record Pointers After LOCKREC

  Following a successful call to LOCKREC, the current-record, next-record, and end-of-file pointers are:

  unchanged.

NEXTFILENAME

The NEXTFILENAME procedure is used to obtain the names of disc files on a designated volume. NEXTFILENAME returns the next file name in alphabetical sequence after the file name supplied as a parameter. The intended use of NEXTFILENAME is in an iterative loop where the file name returned in one call to NEXTFILENAME is used to specify the starting point for the alphabetical search in the subsequent call to NEXTFILENAME. In this manner, a volume's file names are returned to the application process in alphabetical order through succeeding calls to NEXTFILENAME.

The call to the NEXTFILENAME procedure is

```
<error> := NEXTFILENAME ( <file name> )
```

where

<error>, INT — returned

is a file management error number indicating the outcome of the call. Common error number returns are:

- 0 = no error, next file name in alphabetical sequence is returned in <file name>
- 1 = end-of-file, there is no file in alphabetical sequence following the file name supplied in <file name>
- 13 = illegal filename specification

<file name>, INT:ref:12, passed, returned

on the call, is passed the file name from which search for the next file name begins. <file name> on the initial call can be one of the following forms:

<file name[0:11]> = $<volume name><blank fill>
or \<system number><volume name><blank fill>

The form shown above is used to obtain the name of the first file on $<volume name>.

<file name[0:3]> = $<volume name><blank fill>
or \<system number><volume name><blank fill>
<file name[4:11]> = <subvol name><blank fill>

The form shown above is used to obtain the name of the first file in <subvol name> on $<volume name>.

<file name[0:3]> = $<volume name><blank fill>
or \<system number><volume name><blank fill>
<file name[4:7]> = <subvol name><blank fill>
<file name[8:11]> = <disc file name><blank fill>

The form is used to return the name of the next file in alphabetic sequence.

On the return, <file name> is returned the next file name in alphabetical sequence, if any.

condition code settings:

The condition code setting has no meaning following a call to NEXTFILENAME

example

```
fname ':=' [ "$SYSTEM ", 8 * ["   "]];
WHILE NOT (error := NEXTFILENAME ( fname ) ) DO
  BEGIN
    .
    .
  END;
```

OPEN

The OPEN procedure establishes a communication path between an application process and a file. When OPEN completes, a "file number" is returned to the application process. The file number identifies this access to the file in subsequent file management calls.

The call to the OPEN procedure is:

```
CALL OPEN (    <file name>
           ,   <file number>
           , [ <flags>                    ]
           , [ <sync depth>               ]
           , [ <primary file number>      ]
           , [ <primary process id>       ]
           , [ <sequential block buffer>  ]
           , [ <buffer length>            ] )
```

where

<file name>, INT:ref, passed

is an array containing the name of the file to be opened.

<file number>, INT:ref:1, returned

is returned from OPEN and is used to identify the file in subsequent file management calls.

→

<flags>, INT:value, passed

if present, specifies certain attributes of the file. If omitted, all fields are set to zero. The bit fields in the <flags> parameter are defined as follows:

```
<flags>.<1>     = is unused; must be zero

<flags>.<2>     = unstructured access
                    0 = no          1 = yes

<flags>.<3>   } = access mode
<flags>.<4>   }     0 = read/write  1 = read-only
<flags>.<5>   }     2 = write-only

<flags>.<6>     = resident buffering
                    0 = no          1 = yes

<flags>.<7>     = reserved for Link Control Blocks (LCBs)

<flags>.<8>     = open process file no-wait
                    0 = no          1 = yes


<flags>.<9>   } = exclusion mode
<flags>.<10>  }     0 = shared      1 = exclusive
<flags>.<11>  }     3 = protected

<flags>.<12>  } = wait or no-wait i/o
<flags>.<13>  }    0 = wait i/o
<flags>.<14>  }    1 = no-wait i/o
<flags>.<15>  }
```

where

<flags>.<12:15> specifies the maximum number of concurrent no-wait i/o operations that can be in progress on this file at any given time. <flags>.<12:15> = 0 implies "wait i/o". For disc files, only one no-wait operation can be outstanding at one time (i.e., maximum value for <flags>.<12:15> is 1).

<flags>.<9:11> specifies exclusion mode:

0 = shared access
1 = exclusive access
3 = protected access

→

<flags>.<8> = 1 indicates that for process files, the OPEN message is sent no-wait and must be completed by a call to AWAITIO.

<flags>.<6> = 1 specifies resident buffering for unstructured files on Tandem NonStop System only. For Tandem NonStop II System, this bit must be 0. (See Section 4, File Access, for details.) Resident buffering is not permitted with ENSCRIBE file structures.

<flags>.<3:5> specifies access mode:

0 = read/write access
1 = read-only access
2 = write-only access

<flags>.<2> specifies unstructured access regardless of the actual file structure. This field should be set to 0 to provide normal access to the file. (See considerations for details.)

<sync depth>, INT:value, passed

if present, specifies the number of nonretryable (i.e. write) requests whose completion status is to be remembered by the file system. A value of one or greater must be specified to recover from a path failure occurring during a write operation.

<sync depth> also implies the number of write operations the primary process in a primary/backup process pair may perform to this file without intervening checkpoints to its backup process.

If omitted, or zero is specified, internal checkpointing does not occur and disc path failures are not automatically retried by the file system.

The next two parameters are supplied only if the open is by the backup process of a process-pair, the file is currently open by the primary process, and the Checkpointing Facility (described in the GUARDIAN Operating System Programming Manual) is not used.

<primary file number>, INT:value, passed

is the file number returned to the primary process when it opened this file.

<primary process id>, INT:ref, passed

is an array which contains the <process id> of the corresponding primary process. The primary process must already have the file open.

The next two parameters are included if the block buffer for the file is to reside in the application process's data area. Otherwise, the next two parameters are omitted. See section 4 for an explanation of the "Sequential Buffer Option".

Note: The file must be opened with protected or exclusive access if sequential buffering is to be used.

<sequential block buffer>, INT:ref, passed

is an array which the caller is providing for unblocking records to speed sequential processing.

<buffer length>, INT:value, passed

is the length (in bytes) of the <sequential block buffer>. <buffer length> must be greater than or equal to the <data block length> specified at creation for this file and any associated alternate key file(s). If not, or if the file is opened with shared access, the open succeeds but returns a CCG indication (a subsequent call to FILEINFO returns <error> = 5); the application process's sequential buffer is not used; instead, normal system buffering is used. If this parameter is omitted or specified as zero, sequential buffering will not be attempted.

Condition code settings:

```
   <  (CCL) indicates that the OPEN failed (call FILEINFO)
   =  (CCE) indicates that the file opened successfully
   >  (CCG) indicates that the file opened successfully but an
            exceptional condition was detected (call FILEINFO)
```

example

```
   CALL OPEN ( filename, filenum ); ! "wait i/o", exclusion mode
                                    ! = shared, access mode =
                                    ! read/write, sync depth = 0.
   IF < THEN ....                   ! OPEN failed.
```

General Considerations

- How File Numbers are Assigned

  Within a process, the file numbers are unique. The lowest numerical file number is zero (0) and is reserved for $RECEIVE. Remaining file numbers start at one (1). The lowest available file number is always assigned. Once a file is closed, its file number becomes available and a subsequent file open may reuse that file number.

- Maximum Number of Open Files

  The maximum number of files in the system that can be open at any given time depends on the space available for control blocks (ACB's and FCB's). The amount of space available for control blocks is limited only by the physical memory size of the system.

- Multiple Opens by Same Process

  If a given file is opened more than once by the same process, a new ACB is created for each OPEN. This provides logically separate accesses to the same file (a unique <file number> is provided for each OPEN).

- Maximum Opens on Same File

  For disc files, there is no limit on concurrent opens for the same file.

- Maximum Number of Nowait Opens for Same File

  The maximum number of concurrent no-wait operations permitted for an open of a disc file is one (1). Attempting to open a disc file and specifying a value greater than one returns an error indication. A subsequent call to FILEINFO returns <error> 28.

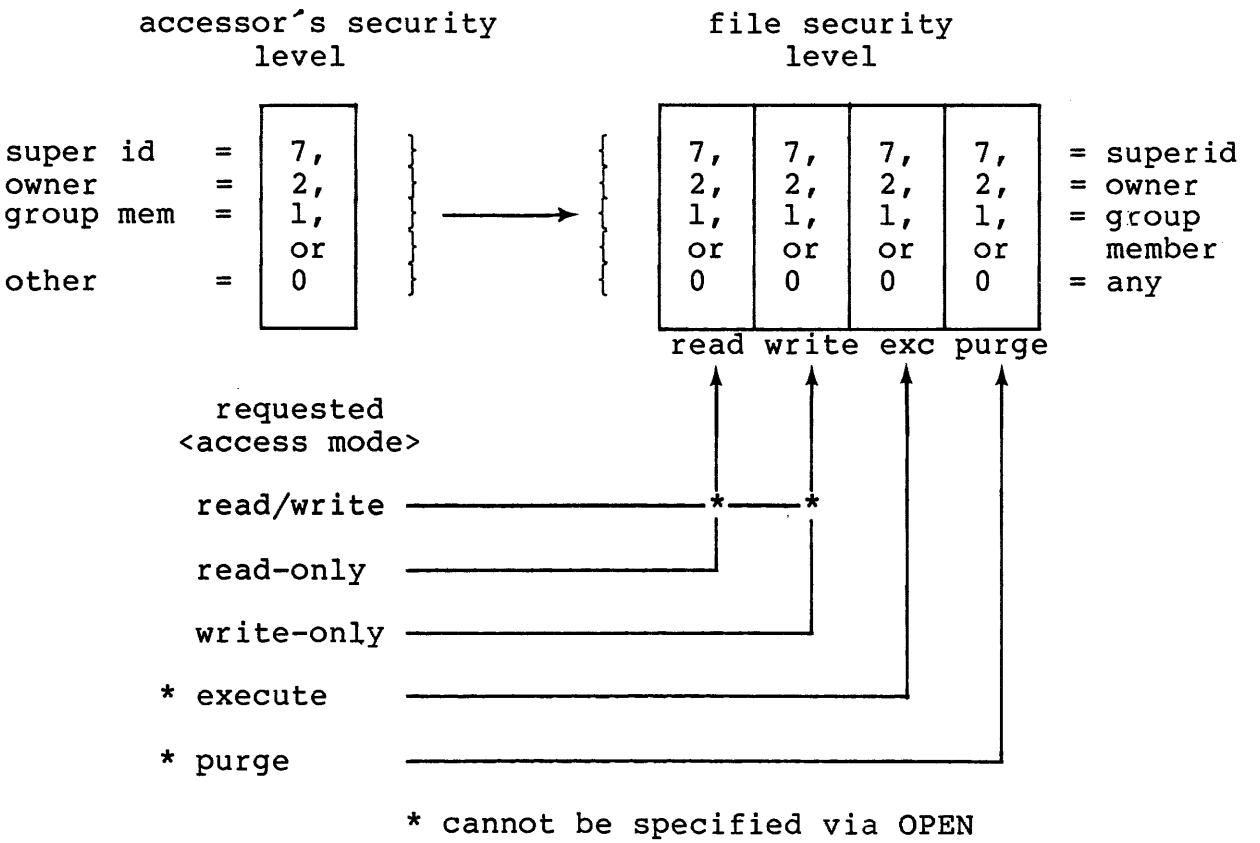
- Errors Returned for No-Wait Files

  See Error Recovery Considerations in Section 4, File Access, for considerations when using "no-wait" i/o.

- File Security Checking on File Open

  When a disc file open is attempted, a file security check takes place. The accessor's (i.e., caller's) security level is checked against the file's security level for the requested access mode. (File security is set via the SETMODE Procedure or the File Utility Program, FUP, SECURE Command.) If the caller's security level is equal-to or higher-than the file's security level for the requested access mode, then the caller passes the security check. If the caller fails the security check, the open fails and a subsequent call to FILEINFO returns <error> 48: security violation.

  The file security checking performed by the file system at open time is illustrated in Figure 3-2, on the following page.



If the accessor's security level is equal-to or higher-than the file's security level for the requested access mode, then the accessor passes the security check

Figure 3-2. File Security Checking

- Exclusion/Access Mode Checking on File Open

  When a file open is attempted, the requested access and exclusion modes are compared to those of any opens already granted for the file. If the attempted open is in conflict with other opens, then the open fails. A subsequent call to FILEINFO returns <error> 12. Table 3-8, on the following page, lists all possible current modes and requested modes; the table indicates whether an open succeeds or fails.

  Note: "Protected" exclusion mode has meaning only for disc files. For other files, specifying "protected" exclusion mode is equivalent to specifying "shared" exclusion mode.

Table 3-8. Exclusion/Access Mode Checking

OPEN ATTEMPTED WITH		FILE CURRENTLY OPEN WITH									
Exclusion Mode		CLOSED	S	S	S	E	E	E	P	P	P
	Access Mode		R/W	R	W	R/W	R	W	R/W	R	W
S	R/W	Y	Y	Y	Y						
S	R	Y	Y	Y	Y				Y	Y	Y
S	W	Y	Y	Y	Y						
E	R/W	Y	ALWAYS FAILS								
E	R	Y									
E	W	Y									
P	R/W	Y		Y							
P	R	Y		Y						Y	
P	W	Y		Y							

Exclusion Mode:
- S = Shareable
- E = Exclusive
- P = Protected

Access Mode:
- R/W = Read/Write
- R = Read only
- W = Write only

Y = Yes, OPEN successful
Blank = No, OPEN fails.

Notes:
- BACKUP opens the file currently being backed-up with R, P.
- BACKUP with "OPEN" option specified opens the file with R, S.
- RESTORE opens the file currently being restored with R/W, E.
- When a program file is running it is opened with the equivalent to R, P.

- REFRESH (CREATE Option) Action

  When a disc file that has the REFRESH option set is opened, file labels are refreshed automatically when the end-of-file pointer is advanced. Depending on the particular application, there may be a significant decrease in processing throughput due to the increased number of disc accesses.

- Partitioned Files

  For partitioned files, there is a separate pair of FCB's for each partition of the file. There is one ACB per accessor (as for single volume files), but this ACB requires more main memory since it contains the information necessary to access all of the partitions, including the location and partial key value for each partition.

Considerations for Structured Files

- Accessing Structured Files as Unstructured Files

  The "unstructured access" option (<flags>.<2>) permits a file to be accessed as an unstructured file. For a file open with this option specified, a data transfer occurs to the position in the file specified by a relative byte address (instead of to the position indicated by a key-field or record number); the number of bytes transferred is that specified in the file management procedure call (instead of the number of bytes indicated by the record format). If a partitioned structured file is opened as an unstructured file, only the first partition is opened. The remaining partitions must be opened individually with separate calls to OPEN (each OPEN specifying unstructured access).

  CAUTION

  Programmers using this option are cautioned that the block format used by ENSCRIBE must be maintained if the file is to ever be accessed again in its structured form.

  The ENSCRIBE block format is described in Appendix C.

- Current State Indicators After OPEN

  Current state indicators following completion of a successful OPEN:

current position	is that of the first record in the file by primary key.
positioning mode	is set to approximate.
compare length	is 0.

  For key-sequenced files, KEYPOSITION must be called after OPEN to establish a position in the file before a subsequent i/o call (READ, READUPDATE, WRITE, etc.) can be made.

  For relative and entry-sequenced files, a READ following an OPEN reads the first record in the file. Subsequent reads without intervening positioning reads the file sequentially through the last record in the file.

Considerations for Unstructured Files

- Multiple OPENs for Single Unstructured File

  If an unstructured disc file is opened by more than one process, separate current-record and next-record pointers are maintained for each opener, but all of the processes share the same end-of-file pointer.

- File Pointers After OPEN

  Following an open to a disc file, the current-record and next-record pointers start out pointing to relative byte address zero and the first data transfer (unless an intervening POSITION is performed) is from that location. The pointers following a successful OPEN are:

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

POSITION

The POSITION procedure is used to position by primary key within relative and entry-sequenced files. For unstructured files, the POSITION procedure specifies a new current position.

For relative and unstructured files, POSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The POSITION procedure is not used with key-sequenced files.

The caller is not suspended because of a call to POSITION.

A call to the POSITION procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the POSITION procedure is:

```
CALL POSITION (  <file number>
              , <record specifier> )
```

where

<file number>, INT:value, passed

identifies the file to be positioned.

<record specifier>, INT(32):value, passed

specifies the new setting for the current-record and next-record pointers:

Relative files: <record specifier> is a four-byte <record number>. -2D specifies that the the next write should occur at an unused record position. -1D specifies that subsequent writes should be appended to the end-of-file location. (-2D and -1D remain in effect until a new <record specifier> is supplied.)

Entry-Sequenced files: <record specifier> is a four-byte <record address>.

Unstructured files: <record specifier> is a four-byte <relative byte address>. -1D specifies that subsequent writes should be appended to the end-of-file location. (-1D remains in effect until a new <record specifier> is supplied.)

condition code settings:

<	(CCL)	indicates that an error occurred (call FILEINFO)
=	(CCE)	indicates that the POSITION was successful
>	(CCG)	no operation, not an unstuctured, relative, or entry-sequenced file

example

```
CALL POSITION ( infile, 1000D );
IF < THEN ....                    ! error occurred
```

Considerations for Relative and Entry-Sequenced Files

- Writing to Entry-Sequenced Files

  Inserts to entry-sequenced files always occur at the end of file.

- Current State Indicators for Structured Files

  Following a successful POSITION to a relative or entry-sequenced file, the current state indicators are:

current position	is that of the record indicated by the <record specifier>.
positioning mode	is approximate.
compare length	is 4.
current primary key value	is set to the value of the <record specifier>.

Considerations for Unstructured Files

- Value of <record specifier> for Unstructured Files

  Unless the unstructured file was created with the ODDUNSTR parameter set, the rba passed in <record specifier> must be an even number. If the ODDUNSTR parameter was set when the file was created, the rba passed in <record specifier> can be either an odd or even value. (The ODDUNSTR parameter is set with <file type>.<12> of the CREATE procedure.)

- Meaning of -2D for Unstructured Files

  Specifying -2D for <record specifier> is equivalent to -1D for unstructured files.

- File Pointers After POSITION

  Following a successful call to POSITION for an unstructured file, the file pointers are:

  ```
  current-record pointer := next-record pointer :=
     if <rba> = -1D then end-of-file pointer else <rba>
  ```

PURGE

The PURGE procedure is used to delete a closed disc file. When PURGE is executed the disc file name is deleted from the volume's directory and any space previously allocated to that file is made available to other files.

The call to PURGE is:

```
CALL PURGE ( <file name> )

where

   <file name>, INT:ref,                                    passed

     is an array containing the name of the disc file to be
     purged.

       To purge a permanent disc file, <file name> must be of
       the form:

         <file name[0:3]>  is $<volume name><blank fill>
                        or \<system number><volume name><blank fill>
         <file name[4:7]>  is <subvol name><blank fill>
         <file name[8:11]> is <disc file name><blank fill>

       To purge a temporary disc file, <file name> must be of the
       form:

         <file name[0:3]>  is $<volume name><blank fill>
                        or \<system number><volume name><blank fill>
         <file name[4:11]> is <temporary file name>


condition code settings:

   <  (CCL) indicates that the PURGE failed (call FILEINFO)
   =  (CCE) indicates that the file was purged successfully
   >  (CCG) indicates that the device is not a disc or that not
            all partitions of a partitioned file were purged
```

→

example

```
CALL PURGE ( oldfilename );
IF < THEN ...                    ! PURGE failed.
```

Considerations

- Error Recovery

  If PURGE fails, the reason for the failure can be determined by calling FILEINFO, passing -1 as the <file number> parameter.

READ

The READ procedure is used to perform sequential reading of a disc file. For key-sequenced, relative, and entry-sequenced files, the READ procedure reads a subset of records in the file. (A subset of records is defined by an access path, positioning mode, and compare length.) For unstructured files, the READ procedure reads records sequentially on the basis of a beginning relative byte address and the lengths of the records read. (After each READ, the current-record pointer is set to the previous next-record pointer and the next-record pointer is set to the previous next-record pointer plus the number of bytes read.)

For key-sequenced, relative, and entry-sequenced files, the first call to READ following a position returns the first record of the subset (i.e., the record at the current position). Subsequent calls to READ without intermediate positioning return successive records in the subset. Following each READ of the subset's records, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an end-of-file indication.

If the READ procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the READ procedure is:

```
CALL READ (    <file number>
           ,   <buffer>
           ,   <read count>
           , [ <count read>  ]
           , [ <tag>         ]  )
```

where

<file number>, INT:value, passed

identifies the file to be read.

<buffer>, INT:ref:*, returned

is an array in the application process where the information read from the file is returned.

→

<read count>, INT:value, passed

is the number of bytes to be read: {0:4096}.

<count read>, INT:ref:1, returned

for wait i/o only, if present, is returned a count of the number of bytes returned from the file into <buffer>.

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the operation completes.

condition code settings:

<	(CCL) indicates that an error occurred (call FILEINFO)
=	(CCE) indicates that the READ was successful
>	(CCG) end-of-file. No more records in this subset

example

```
CALL READ ( filenum, inbuffer, 72 );
IF < THEN ....                          ! READ failed.
```

General Considerations

- Meaning of <count read> for Wait and Nowait I/O Reads

  If a "wait" read is executed, the <count read> parameter indicates the number of bytes actually read.

  If a "no-wait" read is executed, <count read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the i/o operation completes via the <count transferred> parameter of the AWAITIO procedure.

- Reading a Locked File with the Default Locking Mode

  If the default locking mode is in effect when a call to READ is made and the current record or the file is locked but not locked through the <file number> supplied in the call, the caller of READ is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file/record.

  Note that a deadlock condition occurs if a call to READ is made by the process having a file/record locked but not locked via the <file number> supplied to READ.

- Reading a Locked File with the Alternate Locking Mode

  If the alternate locking mode is in effect when a call to READ is made and the current record or the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file/record is locked" error indication (<error> = 73).

- Selecting the Locking Mode

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

Considerations for Structured Files

- Selecting a Subset of Records for Sequential Reads

  The subset of records read by a series of calls to READ is specified through the POSITION or KEYPOSITION procedures.

- Sequential Reads of an Approximate Subset of Records

  If an approximate subset is being read, the first record returned is the one whose key field, as indicated by the current key specifier, contains a value equal to or greater than the current key. Subsequent reading of the subset returns successive records until the last record in the file is read (an end-of-file indication is then returned).

  Sequential reading of an approximate subset in a relative file will skip deleted records.

- Sequential Reads of a Generic Subset of Records

  If generic subset is being read, the first record returned is the one whose key field, as designated by the current key specifier, contains a value equal to the current key for compare length bytes. Subsequent reading of the file returns successive records whose key matches the current key (for compare length bytes). When the current key no longer matches, an end-of-file indication is returned.

  For relative and entry-sequenced files, a generic subset of the primary key is equivalent to an exact subset.

- Sequential Reads of an Exact Subset of Records

  If an exact subset is being read, the only records returned are those whose key field, as designated by the current key specifier, contains a value of exactly compare length bytes and is equal to key. When the current key no longer matches, an end-of-file indication is returned. The exact subset for a key field having a unique value is at most one record.

- Current State Indicators After a Read

  Current state indicators following a successful READ:

current position	is that of the record just read.
positioning mode	is unchanged.
compare length	is unchanged.
current primary key value	is set to the value of the primary key field in the record.

Considerations for Unstructured Files

- Unstructured READs

  For a read from an unstructured disc file, data transfer begins at the position indicated by the next-record pointer.

- How Many Bytes are READ

  If the unstructured file was created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes read is exactly the number of bytes specified with <read count>. If the ODDUNSTR parameter was not set when the file was created, the value of <read count> is rounded up to an even number before the READ is executed.

  The ODDUNSTR parameter is set with <file type>.<12> of the CREATE procedure.

- Determination of <count read> for Unstructured READs

  Following a successful call to READ to an unstructured file, the value returned in <count read> is determined by:

  ```
  <count read> := $MIN ( <read count> , end-of-file pointer
                                       - next-record pointer )
  ```

- File Pointers After a READ

  Following a successful READ to an unstructured file, the file pointers are:

  ```
  CCG := if next-record pointer = end-of-file pointer then 1
                                                    else 0;

  current-record pointer := next-record pointer;

  next-record pointer := next-record pointer + <count read>;
  ```

READLOCK

The READLOCK procedure is used to perform sequential locking and reading of records in a disc file. For key-sequenced, relative, and entry-sequenced files, a subset of the file (defined by the current access path, positioning mode, and compare length) is locked and read with successive calls to READLOCK. For unstructured files, the relative byte address (rba) of the record returned by the READLOCK procedure is locked before the record data is transferred.

For key-sequenced, relative, and entry-sequenced files, the first call to READLOCK following a position (or OPEN) first locks and then returns the first record of the subset. Subsequent calls to READLOCK without intermediate positioning, lock, then return successive records in the subset. Following each read of the subset's records, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an end-of-file indication.

If the READLOCK procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the READLOCK procedure is:

```
CALL READLOCK (    <file number>
               ,   <buffer>
               ,   <read count>
               , [ <count read>  ]
               , [ <tag>         ]  )
```

where

<file number>, INT:value, passed

identifies the file to be read.

<buffer>, INT:ref:*, returned

is an array in the application process where the information read from the file is returned.

<read count>, INT:value, passed

is the number of bytes to be read: {0:4096}.

→

```
<count read>, INT:ref:1,                                    returned

  for wait i/o only, if present, is returned a count of the
  number of bytes returned from the file into <buffer>.


<tag>, INT(32):value,                                       passed

  for no-wait i/o only, if present, is stored by the system,
  then passed back to the application process by the AWAITIO
  procedure when the operation completes.


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the READLOCK was successful
   >  (CCG) end-of-file.  No more records in this subset


example

   CALL READLOCK ( filenum, inbuffer, 72, num^read );
   IF < THEN ....                    ! READLOCK failed.
```

Considerations

- See the considerations for READ.

- How READLOCK Works

  The record locking performed by READLOCK functions identically with that of LOCKREC.

- Locking Records in an Unstructured File

  READLOCK can be used to lock record positions, represented by a relative byte address (rba), in an unstructured file. When sequentially reading an unstructured file with READLOCK, each call to READLOCK first locks the rba stored in the current next-record pointer and then returns record data beginning at the current next-record pointer for <read count> bytes. Following a successful READLOCK, the current-record pointer is set to the previous next-record pointer, and the next-record pointer is set to the previous next-record pointer plus <read count>. This process is repeated for each subsequent call to READLOCK.

READUPDATE

The READUPDATE procedure is used for random processing of records in a disc file. A call to READUPDATE returns the record from the current position in the file. Because READUPDATE is designed for random processing, it cannot be used for successive positioning through a subset of records like the READ procedure. Rather, READUPDATE is intended to be used to read a record after a call to POSITION or KEYPOSITION, possibly in anticipation of a subsequent update through a call to the WRITEUPDATE procedure.

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATE is always to the record described by the exact value of the current key and current key specifier. If such a record does not exist, the call to READUPDATE is rejected with a "record does not exist" error (<error> = 11). (This is unlike sequential processing via the READ procedure, where positioning may be by approximate, generic, or exact key value.)

For unstructured files, data is read from the file beginning at the position of the current-record pointer. A call to READUPDATE typically follows a call to POSITION that sets the current-record pointer to the desired relative-byte-address (rba). The values of the current-record and next-record pointers are not changed by a call to READUPDATE.

If the READUPDATE procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the READUPDATE procedure is:

```
CALL READUPDATE (    <file number>
                 ,   <buffer>
                 ,   <read count>
                 , [ <count read>  ]
                 , [ <tag>         ] )

where

   <file number>, INT:value,                                  passed

     identifies the file to be read.
```

→

```
    <buffer>, INT:ref:*,                                    returned

      is an array where the information read from the file is
      returned.


    <read count>, INT:value,                                  passed

      is the number of bytes to be read {0:4096}.


    <count read>, INT:ref:1,                                returned

      for wait i/o only, if present, is returned a count of the
      number of bytes returned from the file into <buffer>.


    <tag>, INT(32):value,                                     passed

      for no-wait i/o only, if present, is stored by the system,
      then passed back to the application process by the AWAITIO
      procedure when the operation completes.


  condition code settings:

    <  (CCL) indicates that an error occurred (call FILEINFO)
    =  (CCE) indicates that the READUPDATE was successful
    >  (CCG) is not returned by READUPDATE


  example

    CALL READUPDATE ( infile, inbuffer, 72 );
    IF < THEN ....                          ! READUPDATE failed
```

General Considerations

- Calling READUPDATE After READ

  A call to READUPDATE following a call to READ, without intermediate positioning, returns the same record as the READ.

- Meaning of <count read> for Wait and Nowait I/O

  If a "wait" read is executed, the <count read> parameter indicates the number of bytes actually read.

  If a "no-wait" read is executed, <count read> has no meaning and can be omitted. The count of the number of bytes read is obtained when the i/o operation completes via the <count transferred> parameter of the AWAITIO procedure.

- READUPDATE to Locked File with Default Locking Mode

  If the default locking mode is in effect when a call to READUPDATE is made and the current record or the file is locked but not locked through the <file number> supplied in the call, the caller of READUPDATE is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file/record.

  Note that a deadlock condition occurs if a call to READUPDATE is made by the process having a file/record locked but not locked via the <file number> supplied to READUPDATE.

- READUPDATE to Locked File with Alternate Locking Mode

  If the alternate locking mode is in effect when a call to READUPDATE is made and the current record or the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file/record is locked" error indication (<error> = 73).

- Selecting Locking Mode

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

Considerations for Structured Files

- Calling READUPDATE Without Selecting a Specific Record

  If the call to READUPDATE immediately follows a call to KEYPOSITION where a non-unique alternate key is specified, the READUPDATE fails. A subsequent call to FILEINFO returns error 46 (invalid key). However, if an intermediate call to READ[LOCK] is performed, the call to READUPDATE is permitted because a unique record is identified.

- Current State Indicators After READUPDATE

  Current state indicators following a successful READUPDATE:

  unchanged.

Considerations for Unstructured Files

- Unstructured Reads with READUPDATE

  For a read from an unstructured disc file, data transfer begins at the position indicated by the current-record pointer.

- How Many Bytes are Read

  If the unstructured file was created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes read is exactly the number of bytes read specified with <read count>. If the ODDUNSTR parameter was not set when the file was created, the value of <read count> is rounded up to an even value before the READUPDATE is executed.

  The ODDUNSTR parameter is set with <file type>.<12> of the CREATE procedure.

- Determination of <count read> for Unstructured Files

  Following a successful call to READUPDATE to an unstructured file, the value returned in <count read> is determined by:

  ```
  <count read> := $MIN ( <read count> , end-of-file pointer
                                       - next-record pointer )
  ```

- File Pointers After READUPDATE

  Following a successful call to READUPDATE, the current-record pointer and next-record pointer are:

  unchanged.

READUPDATELOCK

The READUPDATELOCK procedure is used for random processing of records in a disc file. A call to READUPDATELOCK locks, then returns the record from the current position in the file. READUPDATELOCK is is intended to be used to read a record after a call to POSITION or KEYPOSITION, possibly in anticipation of a subsequent call to the WRITEUPDATE[UNLOCK] procedure.

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATELOCK is always to the record descibed by the exact value of the current key and current key specifier. If such a record does not exist, the call to READUPDATELOCK is rejected with a "record does not exist" error (<error> = 11).

A call to READUPDATELOCK is functionally equivalent to a call to LOCKREC followed by a call to READUPDATE. However, less system processing is incurred when the READUPDATELOCK Procedure is called rather than when two separate calls are made to LOCKREC and READUPDATE.

If the READUPDATELOCK procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the READUPDATELOCK procedure is:

```
CALL READUPDATELOCK (    <file number>
                     ,   <buffer>
                     ,   <read count>
                     , [ <count read>  ]
                     , [ <tag>         ] )
```

where

<file number>, INT:value, passed

identifies the file to be read.

<buffer>, INT:ref:*, returned

is an array where the information read from the file is returned.

<read count>, INT:value, passed

is the number of bytes to be read {0:4096}.

<count read>, INT:ref:1, returned

for wait i/o only, if present, is returned a count of the number of bytes returned from the file into <buffer>

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the operation completes

condition code settings:

<	(CCL) indicates that an error occurred (call FILEINFO)
=	(CCE) indicates that the READUPDATELOCK was successful
>	(CCG) is not returned from READUPDATELOCK

example

```
CALL READUPDATELOCK ( infile, inbuffer, 72, num^read );
IF < THEN ....                    ! READUPDATELOCK failed.
```

Considerations

- See the considerations for READUPDATE

- How READUPDATELOCK Works

  The record locking performed by READUPDATELOCK functions identically with that of LOCKREC.

REFRESH

The REFRESH procedure is used to write control information contained in File Control Blocks (FCBs), such as the end-of-file pointer, to the associated physical disc volume. (While a file is open, its control information is kept in its main-memory resident FCB; this control information is normally written to the physical volume only when the last process having the file open closes the file.) This procedure or the equivalent Peripheral Utility Program (PUP) REFRESH command should be performed for all volumes prior to a total system shutdown.

The call to the REFRESH procedure is:

```
CALL REFRESH [ ( $<volume name> ) ]

where

   $<volume name>, INT:ref,                                  passed

      specifies a volume whose associated FCB's should be written
      to disc.  $<volume name> can be specified as a full
      twelve-word <file name>; <file name[4:11]> is ignored.

      If omitted, all FCB's for all volumes are written to their
      respective discs.

example

   CALL REFRESH;
```

Consideration

- Calling REFRESH without Specifying <volume name>

  When REFRESH is called without a <volume name>, the error returned is always zero (CCE).

RENAME

The RENAME procedure is used to change the name of an open disc file. If the file is temporary, assigning a new name causes the file to be made permanent.

A call to the RENAME procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the RENAME procedure is:

```
CALL RENAME (  <file number>
             , <new name> )

where

   <file number>, INT:value,                                  passed

     identifies the file to be renamed.


   <new name>, INT:ref,                                       passed

     is an array containing the <file name> to be assigned to the
     disc file, in the following form:

       <file name[0:3]>  is $<volume name><blank fill>
                      or \<system number><volume name><blank fill>
       <file name[4:7]>  is <subvol name><blank fill>
       <file name[8:11]> is <disc file name><blank fill>


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the RENAME was successful
   >  (CCG) the device is not a disc


example

   CALL RENAME ( temp^file, name^array );
   IF < THEN ...                    ! error occurred.
```

Considerations

- Access Security Required for RENAME

  The caller must have purge access to the file for the RENAME to be successful. Otherwise, the RENAME will be rejected with a file management <error> 48, "security violation".

- Volume Name Requirement

  The <volume> specified in <new name> must be the same as the <volume> specified when opening the file.

REPOSITION

The REPOSITION procedure is used to position a disc file to a "saved" position (the positioning information having been saved by a calling the SAVEPOSITION procedure). The REPOSITION procedure passes the positioning block obtained via SAVEPOSITION back to the file system. Following a call to the REPOSITION, the disc file is positioned to the point where it was when SAVEPOSITION was called.

A call to the REPOSITION procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the REPOSITION procedure is:

```
CALL REPOSITION (  <file number>
                 , <positioning block> )

where

   <file number>, INT:value,                                  passed

     identifies the file to be positioned to a "saved" position.


   <positioning block>, INT:ref,                              passed

     indicates a "saved" position to be repositioned to.


condition code settings

   < (CCL) indicates that an error occurred (call FILEINFO)
   = (CCE) indicates that REPOSITION was successful
   > (CCG) file is not a disc file


example

   CALL REPOSITION ( file^num, position^block );
   IF < THEN .....;                                    ! error
```

→

SAVEPOSITION

The SAVEPOSITION procedure is used to save a disc file's current file positioning information in anticipation of a need to return to that position. SAVEPOSITION returns a block of positioning information. This block of information is passed back to the file system in a call to the REPOSITION procedure when it is desired to return to the "saved" position.

A call to the SAVEPOSITION procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the SAVEPOSITION procedure is:

```
CALL SAVEPOSITION (    <file number>
                   ,   <positioning block>
                   , [ <positioning block size> ] )
```

where

<file number>, INT:value, passed

identifies the file whose positioning block is to be obtained.

<positioning block>, INT:ref:*, returned

is returned the positioning block for this file.

<positioning block size>, INT:ref:1, returned

is returned a count of the the number of words in the positioning block.

For unstructured files, the count is 4.

For structured files, the count is calculated by
7 + (<max alt key len> + <pri key len> + 1) / 2

→

```
condition code settings

   < (CCL) indicates that an error occurred (call FILEINFO)
   = (CCE) indicates that SAVEPOSITION was successful
   > (CCG) file is not a disc file

example

   CALL SAVEPOSITION ( file^num, position^block );
   IF < THEN .....;                                  ! error
```

SETMODE

The SETMODE procedure is used to set device-dependent functions.

A call to the SETMODE procedure will be rejected with an error indication if there are any outstanding "no-wait" operations pending on the specified file.

The call to the SETMODE procedure is:

```
CALL SETMODE (    <file number>
              ,   <function>
              , [ <parameter 1>  ]
              , [ <parameter 2>  ]
              , [ <last params>  ] )
```

where

<file number>, INT:value, passed

identifies the file to receive the SETMODE <function>.

<function>, INT:value, passed

is one of the device dependent functions listed in the "SETMODE Functions" table.

<parameter 1>, INT:value, passed

is one of the parameters listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<parameter 2>, INT:value, passed

is one of the parameters listed in the "SETMODE Functions" table. If omitted, the present value is retained.

NOTE

SETMODE Function Table follows SETMODENOWAIT description.

→

```
   <last params>, INT:ref:2,                                  returned

     if present, is returned the previous settings of <parameter
     1> and <parameter 2> associated with the current <function>.
     The format is:

       <last params[0]> = old <parameter 1>
       <last params[1]> = old <parameter 2> (if applicable)


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates that the SETMODE was successful
   >  (CCG) indicates that the SETMODE function is not allowed
            for this device type


examples

   CALL SETMODE ( filenum, 3, 1 );       ! disc verify write, on.
   IF > THEN ...                         ! not a disc.

   CALL SETMODE ( termfnum, 1,,, sec ); ! return transfer mode.
   IF < THEN ...
```

Considerations

- SETMODE Defaults

  The SETMODE settings designated as being "default" are the values that apply when a file is opened (not if a particular <function> is omitted when SETMODE is called).

- Obtaining Current SETMODE Settings Without Changing Their Values

  The value of the current setting associated with a <function> is returned to <last values> without changing the current setting if SETMODE is called and both <parameters> are omitted.

- Requirements for Changing File Security and Ownership

  Set disc file security and set disc file owner will be rejected unless the requestor is the owner of the file or the superid.

SETMODENOWAIT

The SETMODENOWAIT procedure is used to set device-dependent functions in a no-wait manner, on no-wait files.

When the SETMODENOWAIT procedure is used to initiate an operation with a file open with "no-wait" specified, the operation must be completed with a corresponding call to the AWAITIO procedure. The <count transferred> parameter to AWAITIO has no meaning for SETMODENOWAIT completions. The <buffer address> parameter is set to the address of <last params> parameter of SETMODENOWAIT.

The call to the SETMODENOWAIT procedure is:

```
CALL SETMODENOWA~  (    <file number>
                   ,    <function>
                   , [ <parameter 1> ]
                   , [ <parameter 2> ]
                   , [ <last params> ]
                   , [ <tag>         ]  )
```

where

<file number>, INT:value, passed

identifies the file to receive the SETMODENOWAIT <function>.

<function>, INT:value, passed

is one of the device-dependent functions listed in the "SETMODE Functions" table.

<parameter 1>, INT:value, passed

is one of the <parameter 1> values listed in the "SETMODE Functions" table. If omitted, the present value is retained.

<parameter 2>, INT:value, passed

is one of the <parameter 2> values listed in the "SETMODE Functions" table. If omitted, the present value is retained.

→

```
    <last params>, INT:ref:2,                                  returned

      if present, is returned the previous settings of
      <parameter 1> and <parameter 2> associated with the
      current <function>.

      The format is:

        <last params[0]> = old <parameter 1>
        <last params[1]> = old <parameter 2> (if applicable)


    <tag>, INT(32):value,                                        passed

      for no-wait i/o only, if present, is stored by the system,
      then passed back to the application process by the AWAITIO
      procedure when the operation completes.


  condition code settings:

     <  (CCL) indicates that an error occurred (call FILEINFO)
     =  (CCE) indicates that the SETMODENOWAIT was successful
     >  (CCG) indicates that the SETMODENOWAIT function is
              not allowed for this device type
```

Considerations

- SETMODENOWAIT Completion

  AWAITIO must be used to complete the call when <file number> is opened with a wait-depth greater than 0. For files with a wait-depth equal to zero, a call to SETMODENOWAIT is a waited operation and performs just as a call to SETMODE.

- <last params> Returned From AWAITIO

  The <buffer> parameter of AWAITIO is set to @<last params>, and the count is undefined.

- SETMODE and SETMODENOWAIT Functions

  A list of SETMODE functions, <parameter 1>, and <parameter 2> settings is shown in Table 3-9, on the following page.

Table 3-9. SETMODE and SETMODENOWAIT Functions

<function>

1 = disc, set file security

<parameter 1>

.<0> = 1, for program files only. Set accessor's id to program file's id when program file is run

.<4:6>, id allowed for read,
.<7:9>, id allowed for write,
.<10:12>, id allowed for execute,
.<13:15>, id allowed for purge,

0 = any local id
1 = member of owner's group
2 = owner
4 = any network user (local or remote)
5 = member of owner's community
6 = local or remote user having same id as owner
7 = local super id only

(see GUARDIAN Programming Manual, Section 8, "Security", for an explanation of local and remote users, communities, etc.)

<parameter 2> is not used

2 = disc, set file owner id

<parameter 1>.<0:7> = group id
.<8:15> = user id

<parameter 2> is not used

3 = disc, set verify write

<parameter 1> = 0, means off (default setting)
= 1, means on

<parameter 2> is not used

→

Table 3-9. SETMODE and SETMODENOWAIT Functions (cont.)

```
 4 = disc, set lock mode

     <parameter 1> = 0, default mode, process will be suspended
                        when lock or read is attempted
                   = 1, alternate mode, lock or read attempt will
                        be rejected with "file is locked" error
                        (<error> = 73)

     <parameter 2> is not used


57 = disc, set serial or parallel writes (overrides SYSGEN setting
                                           for this file)

     <parameter 1> = 1, serial  writes
                   = 2, parallel writes

     <parameter 2> is not used
```

UNLOCKFILE

The UNLOCKFILE procedure is used to unlock a disc file and any records in that file that are currently locked by the caller. Unlocking a file allows other processes to access the file. If any processes are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue). If the process granted access is waiting to lock the file, it is granted the lock (which excludes other process from accessing the file) and resumes processing. If the process granted access is waiting to read the file, its read is processed by the file system.

If the UNLOCKFILE procedure is being used to initiate an operation with a file opened with "no-wait i/o specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the UNLOCKFILE procedure is:

```
CALL UNLOCKFILE (    <file number>
                , [ <tag>         ] )
```

where

<file number>, INT:value, passed

identifies the file to be unlocked.

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the unlock operation completes.

condition code settings:

<	(CCL) indicates that an error occurred (call FILEINFO)
=	(CCE) indicates that the UNLOCKFILE was successful
>	(CCG) file is not a disc file

example

```
CALL UNLOCKFILE ( filenum );
IF < THEN .....                  ! error occurred.
```

UNLOCKREC

The UNLOCKREC procedure is used to unlock a record currently locked by the caller. UNLOCK unlocks the record at the current position, allowing other processes to access that record. If any processes are queued in the locking queue for the record, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue). If the process granted access is waiting to lock the record, it is granted the lock (which excludes other process from accessing the record) and resumes processing. If the process granted access is waiting to read the record, its read is processed by the file system.

If the UNLOCKREC procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

```
CALL UNLOCKREC (    <file number>
                , [ <tag>         ] )
```

where

<file number>, INT:value, passed

identifies the file containing the record to be unlocked.

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the unlock operation completes.

condition code settings:

<	(CCL) indicates that an error occurred (call FILEINFO)
=	(CCE) indicates that the UNLOCKREC was successful
>	(CCG) file is not a disc file

example

```
CALL UNLOCKREC ( filenum );
IF < THEN .....                  ! error occurred.
```

Considerations

- Calling UNLOCKREC After KEYPOSITION

  If the call to UNLOCKREC immediately follows a call to KEYPOSITION where a non-unique alternate key is specified, the UNLOCKREC fails. A subsequent call To FILEINFO returns error 46 (invalid key). However, if an intermediate call to READ[LOCK] is performed, the call to UNLOCKREC is permitted.

- Unlocking Several Records

  If several records need to be unlocked, the UNLOCKFILE Procedure can be called to unlock all records currently locked by the caller (rather than unlocking the records through individual calls to UNLOCKREC).

- Current State Indicators After UNLOCKREC

  For key-sequenced, relative, and entry-sequenced files, the current state indicators following a successful UNLOCKREC are:

  unchanged.

- File Pointers After UNLOCKREC

  For unstructured files, the current-record pointer and the next-record pointer are:

  unchanged.

WRITE

The WRITE operation is used to insert a new record into a file in the position designated by the file's primary key:

- For key-sequenced files, the record is inserted in the position indicated by the value in its primary key field.
- For relative files, following an OPEN or an explicit positioning by its primary key, the record is inserted in the designated position. Subsequent writes without intermediate positioning insert records in successive record positions.

  If -2D is specified in a preceding positioning, the record is inserted in an available record position in the file.

  If -1D is specified in a preceding positioning, the record is inserted following the last record currently existing in the file.
- For entry-sequenced files, the record is inserted following the last record currently existing in the file.
- For unstructured files, the record is inserted at the position indicated by the current value of the next-record pointer.

If the WRITE procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the WRITE procedure is:

```
CALL WRITE (    <file number>
            ,   <buffer>
            ,   <write count>
            , [ <count written> ]
            , [ <tag>           ] )
```

where

<file number>, INT:value, passed

identifies the file to be written.

<buffer>, INT:ref, passed

is an array containing the information to be written to the file.

→

```
   <write count>, INT:value,                                    passed

     is the number of bytes to be written: {0:4096}.

       For key-sequenced and relative files, 0 is illegal;
       for entry-sequenced files, 0 denotes an empty record.


   <count written>, INT:ref:1,                                  returned

     for wait i/o only, if present, is returned a count of the
     number of bytes written to the file.


   <tag>, INT(32):value,                                        passed

     for no-wait i/o only, if present, is stored by the system,
     then passed back to the application process by the AWAITIO
     procedure when the write operation completes.


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates the the WRITE was successful
   >  (CCG) is not returned by WRITE


example

   CALL WRITE ( outfile, outbuffer, 72 );
   IF < THEN ....                    ! error occurred.
```

General Considerations

- Meaning of <count written> for Wait and Nowait I/O

  If a "wait" write is executed, the <count written> parameter indicates the number of bytes actually written.

  If a "no-wait" write is executed, <count written> has no meaning and can be omitted. The count of the number of bytes written is obtained when the i/o completes via the <count transferred> parameter of the AWAITIO procedure.

- Error 73: File is Locked

  If a call to WRITE is made and the file is locked but not locked through the <file number> supplied in the call, the call is rejected with a "file is locked" error indication (<error> = 73).

Considerations for Structured Files

- Inserting Records into Relative and Entry-sequenced Files

  If the insert is to a relative or entry-sequenced file, the file must be positioned currently via its primary key. Otherwise, the WRITE fails and a subsequent call to FILEINFO returns error 46 (invalid key specified).

- Error 10: Record Already Exists

  If the insert is to a key-sequenced or relative file and the record already exists, WRITE fails and a subsequent call to FILEINFO returns error 10 (duplicate record).

- Current State Indicators After WRITE

  Current state indicators following a successful WRITE:

positioning mode	is unchanged.
compare length	is unchanged.

  - For key-sequenced files

current position	is unchanged
current primary key value	is unchanged.

  - For relative and entry-sequenced files

current position	is that of the record just inserted
current primary key value	is set to the value of the record's primary key.

Considerations for Unstructured Files

- Unstructured WRITEs

  If the write is to an unstructured disc file, data is transferred to the record location specified by the next-record pointer. The next-record pointer is updated to point at the record following the record written.

- How Many Bytes are Written

  If an unstructured file was created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes written is exactly the number of bytes specified with <write count>. If the ODDUNSTR parameter was not set when the file was created, the value of <write count> is rounded up to an even number before the WRITE is executed.

  The ODDUNSTR parameter is set with <file type>.<12> of the CREATE procedure.

- File Pointers After WRITE

  Following a successful WRITE to an unstructured file, the file pointers are:

```
current-record pointer := next-record pointer;
next-record pointer := next-record pointer + <count written>;
end-of-file pointer := max ( end-of-file pointer,
                             next-record pointer);
```

WRITEUPDATE

The WRITEUPDATE procedure is used for random and sequential processing of records in a disc file. WRITEUPDATE has two functions:

- Alter the contents of the record at the current position.
- Delete the record at the current position in a key-sequenced or relative file.

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means that positioning for WRITEUPDDATE is always to the record described by the exact value of the current key and current key specifier. If such a record does not exist, the call to WRITEUPDATE is rejected with a "record does not exist" error (<error> = 11).

For unstructured files, data is written in the position indicated by the current-record pointer. A call to WRITEUPDATE for an unstructured file typically follows a call to POSITION, READ, or READUPDATE procedures. The current-record and next-record pointers are not changed by a call to WRITEUPDATE.

If the WRITEUPDATE procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the WRITEUPDATE procedure is:

```
CALL WRITEUPDATE (    <file number>
                  ,   <buffer>
                  ,   <write count>
                  , [ <count written> ]
                  , [ <tag>           ] )
```

where

<file number>, INT:value, passed

identifies the file to be written.

<buffer>, INT:ref, passed

is an array containing the information to be written to the file.

→

```
<write count>, INT:value,                                    passed

  is the number of bytes to be written to the file: {0:4096}.

    For key-sequenced and relative files, 0 means delete the
    record.

    For entry-sequenced files, 0 is illegal.


<count written>, INT:ref:1,                                  returned

  for wait i/o only, if present, is returned a count of the
  number of bytes written to the file.


<tag>, INT(32):value,                                        passed

  for no-wait i/o only, if present, is stored by the system,
  then passed back to the application process by the AWAITIO
  procedure when the write operation completes


condition code settings:

   <  (CCL) indicates that an error occurred (call FILEINFO)
   =  (CCE) indicates the the WRITEUPDATE was successful
   >  (CCG) is not returned by WRITEUPDATE


example

   CALL WRITEUPDATE ( outfile, outbuffer, 512 );
   IF = THEN ....                                  ! successful.
```

General Considerations

- Meaning of <count written> for Wait and Nowait I/O

  If a "wait" write is executed, the <count written> parameter indicates the number of bytes actually written.

  If a "no-wait" write is executed, <count written> has no meaning and can be omitted. The count of the number of bytes written is obtained when the i/o completes via the <count transferred> parameter of the AWAITIO procedure.

- Calling WRITEUPDATE after Calling READ

  A call to WRITEUPDATE following a call to READ, without intermediate positioning, updates the record just read.

- Deleting Locked Records

  Deletion of a locked record implicitly unlocks that record.

Considerations for Structured Files

- Error 73: File/Record is Locked

  If a call to WRITEUPDATE is made and the current record or the file is locked but not through the <file number> supplied in the call, the call is rejected with a "file/record is locked" error indication (<error> = 73).

- Calling WRITEUPDATE After KEYPOSITION

  If the call to WRITEUPDATE immediately follows a call to KEYPOSITION where a non-unique alternate key is specified as the access path, the WRITEUPDATE fails. A subsequent call to FILEINFO returns error 46 (invalid key). However, if an intermediate call to READ[LOCK] is performed, the call to WRITEUPDATE is permitted because a unique record is identified.

- Specifying <write count> for Entry-sequenced Files

  For entry-sequenced files, the value of <write count> must match exactly the <write count> value specified when the record was originally inserted into the file.

- Changing Primary Key of Key-sequenced Record

  An update to a record in a key-sequenced file may not alter the value of the primary key field. Changing the primary key field must be done by deleting the old record (WRITEUPDATE with <write count> = 0) and inserting a new record with the key field changed (WRITE).

- Current State Indicators After WRITEUPDATE

  Current state indicators following a successful WRITEUPDATE:

  unchanged.

Considerations for Unstructured Files

- WRITEUPDATE to an Unstructured File

  If the write is to an unstructured disc file, data is transferred to the record location specified by the current-record pointer.

- How Many Bytes are Written

  If the unstructured file was created with the ODDUNSTR (odd unstructured file) parameter set, the number of bytes written is exactly the number of bytes specified with <write count>. If the ODDUNSTR parameter was not set when the file was created, the value of <write count> is rounded up to an even number before the WRITEUPDATE is executed.

  The ODDUNSTR parameter is set with <file type>.<12> of the CREATE procedure.

- File Pointers Following a Successful WRITEUPDATE

  Following a successful WRITEUPDATE to an unstructured file, the current-record and next-record pointers are:

  unchanged.

WRITEUPDATEUNLOCK

The WRITEUPDATEUNLOCK procedure is used for random processing of records in a disc file. WRITEUPDATEUNLOCK has two functions:

- Alter, then unlock the contents of the record at the current position.
- Delete, the record at the current position in a key-sequenced or relative file.

For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means that positioning for WRITEUPDATEUNLOCK is always to the record descibed by the exact value of the current key and current key specifier. If such a record does not exist, the call to WRITEUPDATEUNLOCK is rejected with a "record does not exist" error (<error> = 11).

For unstructured files, data is written in the position indicated by the current-record pointer. A call to WRITEUPDATEUNLOCK for an unstructured file typically follows a call to POSITION or READUPDATE. The current-record and next-record pointers are not changed by a call to WRITEUPDATEUNLOCK.

A call to WRITEUPDATEUNLOCK is equivalent to a call to WRITEUPDATE followed by a call to UNLOCKREC. However, less system processing is incurred if the WRITEUPDATEUNLOCK Procedure is called instead of the separate calls to WRITEUPDATE and UNLOCKREC.

If the WRITEUPDATEUNLOCK procedure is being used to initiate an operation with a file opened with "no-wait i/o" specified, the operation must be completed with a corresponding call to the AWAITIO procedure.

The call to the WRITEUPDATEUNLOCK procedure is:

```
CALL WRITEUPDATEUNLOCK (    <file number>
                        ,   <buffer>
                        ,   <write count>
                        , [ <count written> ]
                        , [ <tag>           ] )
```

where

<file number>, INT:value, passed

identifies the file to be written.

→

<buffer>, INT:ref, passed

is an array containing the information to be written to the file.

<write count>, INT:value, passed

is the number of bytes to be written to the file: {0:4096}.

For key-sequenced and relative files, 0 means delete the record.

For entry-sequenced files, 0 is illegal.

<count written>, INT:ref:1, returned

for wait i/o only, if present, is returned a count of the number of bytes written to the file.

<tag>, INT(32):value, passed

for no-wait i/o only, if present, is stored by the system, then passed back to the application process by the AWAITIO procedure when the write operation completes.

condition code settings:

```
<   (CCL) indicates that an error occurred (call FILEINFO)
=   (CCE) indicates the the WRITEUPDATEUNLOCK was successful
>   (CCG) is not returned by WRITEUPDATEUNLOCK
```

example

```
CALL WRITEUPDATEUNLOCK ( outfile, outbuffer, 72, num^written );
IF = THEN ....                    ! successful.
```

Considerations

- See the considerations for WRITEUPDATE

- How WRITEUPDATEUNLOCK Works

  The record unlocking performed by WRITEUPDATEUNLOCK functions identically to that of UNLOCKREC.

SECTION 4

ENSCRIBE FILE ACCESS

Topics discussed in this section are:

- File Open
- Access Rules for Structured Files
- Access Rules for Unstructured Files
- Considerations for both Structured and Unstructured Files
  - File and Record Locking
  - Verify Write
  - Refresh
  - Programmatic Extent Allocation and Deallocation
- Error Recovery Considerations
  - For key-sequenced files
  - For files having alternate keys
  - For partitioned files
- Access Examples

File Creation is discussed in section five, "ENSCRIBE File Creation", and in the GUARDIAN Language and Utilities Manual, "FUP Program".

FILE OPEN

The following should be taken into consideration when opening an ENSCRIBE file.

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

- If the file is partitioned, all partitions are automatically opened when the first partition is opened. If one of the partitions cannot be opened, access to the file is still granted. However, a CCG (warning indication) is returned from OPEN. The FILEINFO procedure can then be called to obtain the file management error number, and the FILERECINFO procedure can be called to obtain the number of the partition which did not open.

- Individual partitions cannot be opened separately unless "unstructured access", OPEN <flags>.<2> = 1, is specified. See OPEN Considerations for details on unstructured access.

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

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

ACCESS RULES FOR STRUCTURED FILES

- Sequential Processing

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

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

The first call to READ[LOCK] following file open or a positioning operation, reads the record, if any, at the current position. Subsequent calls to READ[LOCK], without intermediate positioning, return successive records, if any, in the designated subset.

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

Following each call to READ[LOCK], the position of the record returned becomes the current position.

- Random Processing

  The update procedures, READUPDATE[LOCK] and WRITEUPDATE[UNLOCK], are used for random processing. 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), a "record not found" error (<error> = 11) is returned.

  WRITEUPDATE[UNLOCK] cannot be used to alter a record's primary key. If this is to be done, the record must first be deleted, then inserted (i.e., WRITE) using the new value of the primary key.

  If update or lock is attempted immediately following a call to KEYPOSITION where a non-unique alternate key is specified, the update or lock fails with an "invalid key" error (<error> = 46). However, if an intermediate call to READ[LOCK] is performed, the update or lock is permitted.

- Insert

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

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

  An insert of an empty record (i.e., <write count> = 0) is not valid for key-sequenced and relative files, but is valid for entry-sequenced files.

The length of a record to be inserted must be less than or equal to the the record length defined for the file. If not, the insertion is not performed and an "invalid count" error (<error> = 21) is returned.

- Delete

  A delete operation (i.e., WRITEUPDATE[UNLOCK], <count> = 0) always applies to the current position in a file.

- Alternate Keys

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

- Current Position

  Current position is subject to change only following a call to

READ[LOCK]	from any structured file
WRITE	to a relative or entry-sequenced file
KEYPOSITION	for key-sequenced, relative, and entry-sequenced files
POSITION	for relative, entry-sequenced, and unstructured files

  Following a call to READ[LOCK], the current position becomes that of the record just read. Following a call to WRITE for a relative or entry-sequenced file, the current position becomes that of the record just written.

- Current Key Value

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

- Current Primary Key Value

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

  ```
  READ[LOCK]  from any structured file
  WRITE       to a relative or entry-sequenced file,
  KEYPOSITION by primary key for key-sequenced files
  POSITION    by primary key for relative and entry-sequenced files
  ```

  operation.

- End-Of-File Pointer

  Associated with each disc file is an end-of-file pointer. The end-of-file pointer contains the relative byte address of the first byte of the next available block. The end-of-file pointer is advanced automatically when appending to a file each time a new block is allocated at the end-of-file.

  The system maintains the working copy of a file's end-of-file pointer in the File Control Blocks (FCB) that are in both of the two system processes that control the associated disc volume. A file's end-of-file address is physically written on disc every time one of the following events occur:

  - the file is created
  - an extent is allocated for the file
  - a CONTROL operation is performed for the file
  - the last accessor closes the file
  - the REFRESH procedure is called for the file
  - the PUP REFRESH command is executed for the file's volume
  - every time the eof is changed (if the auto-refresh option is in effect)

  The auto-refresh option can be specified when the file is first created. And even if a file is created without the auto-refresh option specified, it can be modified at some later date to include the auto-refresh option.

  When creating a file with the CREATE procedure, the auto-refresh option is specified by setting <file type>.<12> to a one. When creating a file with FUP, the auto-refresh option is specified with the SET REFRESH command. For files created without the auto-refresh option, the option can specified at any time with the FUP ALTER REFRESH command.

  Considerations regarding when and how frequently files should be refreshed are given under Refresh, later in this section.

Sequential Buffer Option

A process can optionally specify (at file open time) that an array in the process's data area be used by the file system for record deblocking. The advantage to using this option is that instead of requesting each record from an i/o process (which results in an interprocess message being sent, an environment switch, and, possibly waiting to obtain space from system data space), an entire block is returned from the i/o process and stored in the process's data area. Once a block is in the application process's data area, subsequent accesses to records within that block require no disc accesses and no environment changes.

Note, however, that this option is meaningful only for sequential reading of a file. Random reading and any writing will not make use of the sequential buffer.

If sequential block buffering is to be used, the file must be opened with exclusion mode = "protected" or "exclusive". The length specified to OPEN for the sequential block buffer must be greater than or equal to the <data block length> specified for the file and any associated alternate key files at creation. If these criteria are not met, the open succeeds but returns a CCG indication (a subsequent call to FILEINFO will return <error> = 5); the application process's sequential buffer is not used; instead, normal system buffering is used.

Example:

```
INT .seq^buffer [ 0:2047 ];  ! sequential block buffer.
  .
  .
flags := %2060; ! read-only, protected, wait i/o.
CALL OPEN ( filename , fnum , flags ,,,, seq^buffer , 4096 );
IF < THEN ! open failed.
       .
       .
ELSE
IF > THEN
  BEGIN   ! open successful.
    CALL FILEINFO ( fnum , error );
    IF error = 5 THEN ! sequential buffer request rejected.
       .
       .
  END
! open successful.
```

The file is then read sequentially in the normal manner.

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

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

ACCESS RULES FOR UNSTRUCTURED FILES

Communication is established with an unstructured disc file through the ENSCRIBE OPEN procedure.

For example, to establish communication with a permanent unstructured disc file, the following could be written in a source program:

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

CALL OPEN ( file^name, file^num,, 1 ); ! wait i/o,
                                       ! share access,
                                       ! read/write access,
                                       ! sync depth = 1.
IF <> THEN ...
```

The file is identified to other ENSCRIBE procedures by "file^num".

Access to an unstructured disc file is terminated using the ENSCRIBE CLOSE procedure:

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

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

```
CALL OPEN ( temp^file, temp^fnum,, 1 ); ! wait i/o,
                                        ! share access,
                                        ! read/write access,
                                        ! sync depth = 1.

IF <> THEN ...
```

Other ENSCRIBE procedures access the temporary file by using "temp^fnum".

Access to a temporary unstructured disc file is terminated and the file is purged using the CLOSE procedure:

```
  .
CALL CLOSE ( temp^fnum );
  .
```

deletes the temporary file from the volume "$VOL2".

If the application program does not want the temporary file purged at close time, the temporary file can be made permanent by use of the file management RENAME procedure:

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

renames the temporary file, making it permanent. Note that the volume name supplied to RENAME must be the same as that used when the temporary file was created.

Relative Byte Addressing and File Pointers

Data in an unstructured disc 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 a file; the first byte in a file is at rba zero.

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

1. A next-record pointer containing the relative byte address of the location where the next disc transfer, due to a READ or WRITE, begins.

2. A current-record pointer containing the relative byte address of the location just read or written and is the address where a disc transfer due to a READUPDATE or WRITEUPDATE begins.

3. An end-of-file pointer containing the relative byte address of the next byte after the last significant data byte in a file. (If the file was created without the ODDUNST parameter set, the end-of-file pointer is always rounded up to an even number.) The end-of-file pointer is incremented automatically when data is appended to the end-of-file (WRITE). It can be set explicitly by calls to the POSITION and CONTROL procedures.

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

A single end-of-file pointer, however, is associated with all opens of a given unstructured disc file. This permits data to be appended to the end-of-file by several different accessors. The end-of-file pointer resides in the unstructured disc file's File Control Block in the disc i/o process environment. A file's end-of-file pointer value is copied from the file label on disc when the file is opened and is not already open.

An unstructured file's end-of-file address is physically written on disc every time one of the following events occur:

- the file is created
- an extent is allocated for the file
- a CONTROL operation is performed for the file
- the last accessor closes the file
- the REFRESH procedure is called for the file
- the PUP REFRESH command is executed for the file's volume
- every time the eof is changed (if the auto-refresh option is in effect)

The auto-refresh option can be specified when the file is first created. And even if a file is created without the auto-refresh option specified, it can be modified at some later date to include the auto-refresh option.

When creating a file with the CREATE procedure, the auto-refresh option is specified by setting <file type>.<12> to a one. When creating a file with FUP, the auto-refresh option is specified with the SET REFRESH command. For files created without the auto-refresh option, the option can specified at any time with the FUP ALTER REFRESH command.

A summary of unstructured disc file pointer action is given in Table 4-1, on the following page.

Table 4-1. File Pointer Action

```
CREATE

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

OPEN (first)

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

OPEN (any)

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

READ

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

WRITE

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

READUPDATE

   file pointers are unchanged

WRITEUPDATE

   file pointers are unchanged

CONTROL (write end-of-file)

   end-of-file pointer := next-record pointer;
   file label end-of-file pointer := end-of-file pointer;
```

Table 4-1. File Pointer Action (cont'd)

CONTROL (purge data)

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

CONTROL (allocate/deallocate extents)

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

POSITION

current-record pointer := next-record pointer := <rba>;

CLOSE (last)

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

where

<count> is the specified transfer count. If the file was created with the ODDUNST parameter set, the value specified for <count> is the number of bytes transferred. If the file was created but the ODDUNST parameter was not set, <count> is rounded-up to an even number before the data transfer takes place.

Sequential Access

READS and WRITES increment the next-record pointer by the number of bytes transferred, therefore automatic sequential access to the file is provided. (If the file was created with the ODDUNST parameter set, the number of bytes transferred and the amount the pointers are incremented is exactly the number of bytes specified with <write count> or <read count>. If the ODDUNST parameter was not set when the file was created, the values of <write count> and <read count> are rounded up to an even number before the transfer takes place and the file pointers are incremented by the rounded up value.)

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

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

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

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

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

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

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

For example, an unstructured file is written on disc, the end-of-file location is at relative byte 4,096. Sequential reads of 512 bytes are executed, starting at relative location 0.

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

Reads one through eight each transfer 512 bytes into "buffer", return "num^read" = 512, and the condition code indicator set to CCE (operation successful).

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

An example of encountering eof is shown in Figure 4-1 below.

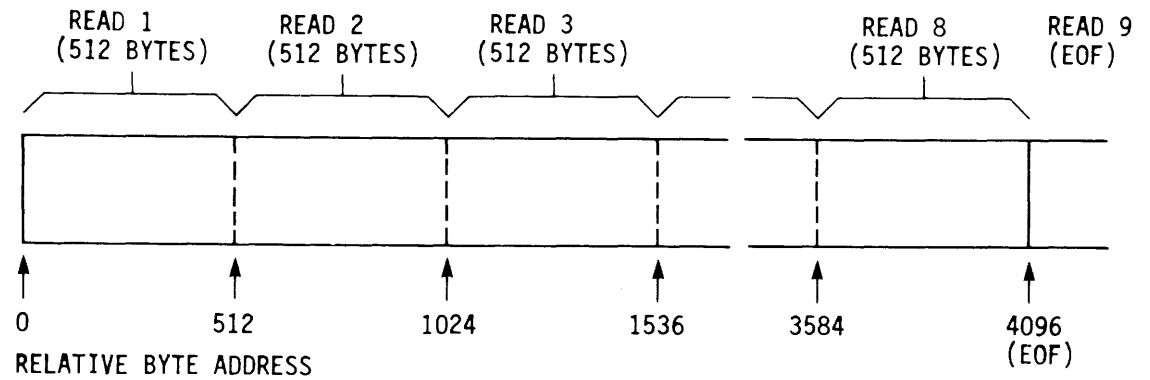


Figure 4-1. Example of Encountering EOF

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

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

In this case, reads one through ten each transfer 400 bytes into "buffer", return "num^read" = 400, and set the condition code indicator to CCE (operation successful).

Read eleven transfers 96 bytes into "buffer", returns "num^read" = 96, and sets the condition code indicator to CCE.

The next read fails and sets the condition code indicator to CCG. This situation is illustrated in Figure 4-2 below.

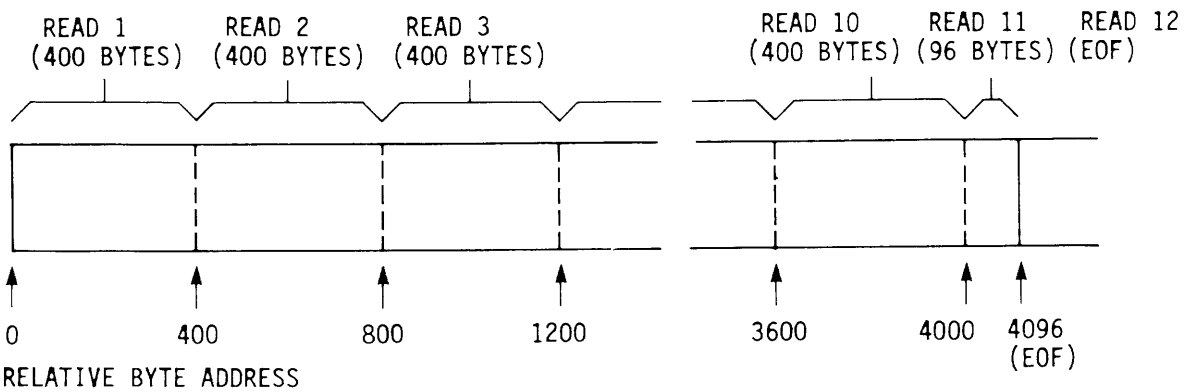


Figure 4-2. Example of Encountering EOF (Short Read)

Random Access

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

For example, to update data in an unstructured disc file at relative byte address 81,920, the following sequence of calls could be made:

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

The call to POSITION sets the next-record and current-record pointers to relative byte 81,920.

The call to READUPDATE transfers 512 bytes from the file to "buffer" starting at relative byte 81,920. Following the read, the next-record and current-record pointers are unchanged.

The WRITEUPDATE procedure replaces the just-read data with new data in the same location on disc. The the file system transfers 512 bytes from "buffer" to the file at relative byte 81,920.

Appending to End-of-File

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

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

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

Then a subsequent WRITE appends 512 bytes to the end of the file:

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

The file system transfers 512 bytes from "buffer" to the current end of file location (131,072). The next-record and end-of-file pointers now point to relative byte 131,584; the current-record pointer points to relative byte 131,072; the next-record pointer still contains -1D so that a subsequent write also appends to end-of-file.

An example of using POSITION for both random access and appending to the end of an unstructured disc file is shown in Figure 4-3 below.

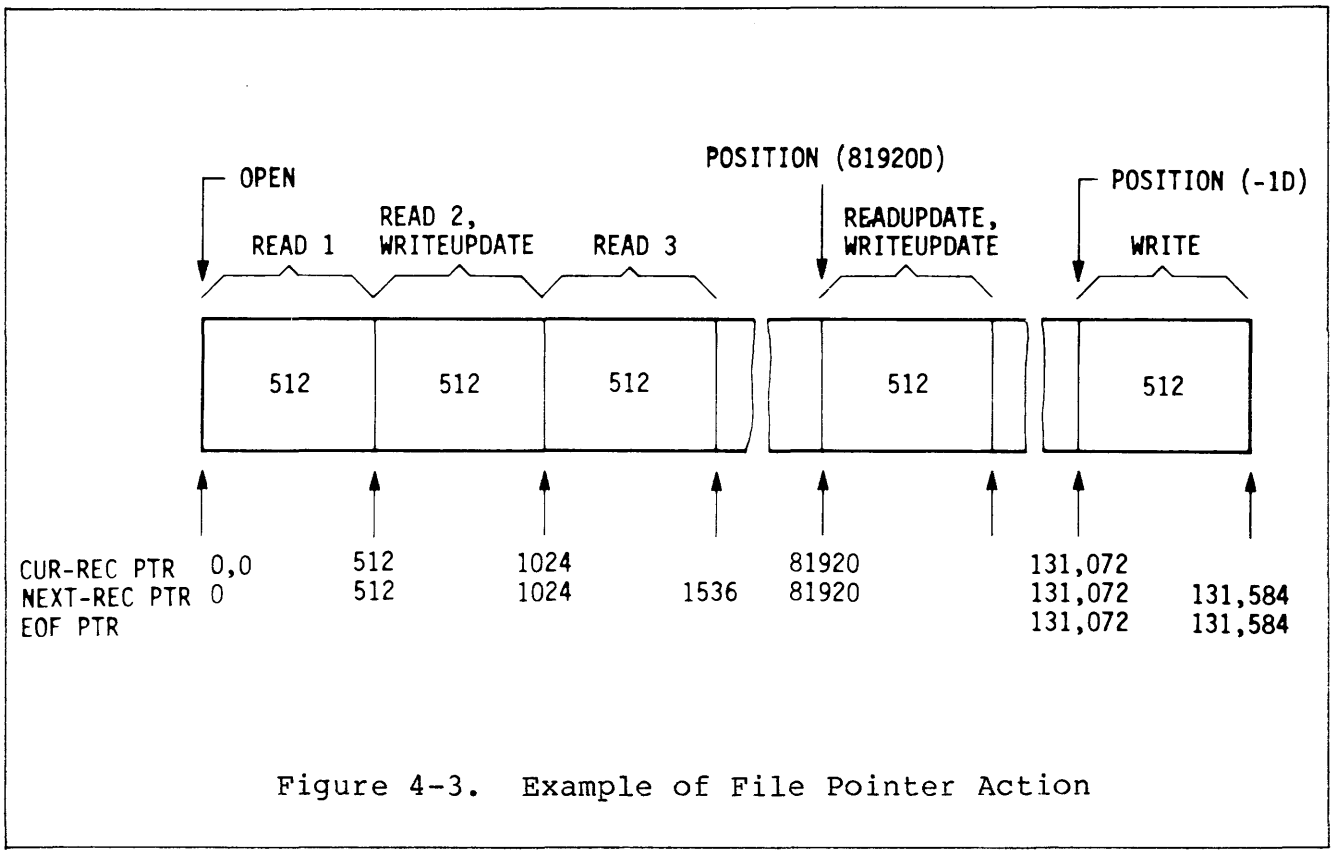


Figure 4-3. Example of File Pointer Action

Disc Sectors

Data is stored on disc in physical locations called sectors. Each sector contains 512 bytes. Any disc read or write operation, regardless of number of bytes transferred, involves at least one sector and can involve as many as eight sectors.

It is most efficient for the file system to transfer whole sectors of information. This means that the most efficient transfers are multiples of 512 bytes on 512 byte boundaries (a file always starts on a sector boundary).

For example, a disc operation transfers 512 bytes of information to an unstructured disc file starting at relative address 512.

```
CALL WRITE ( file^a, array, 512, ... );
```

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

The following example illustrates file system action for a write that crosses sector boundaries. A WRITE of 200 bytes is performed starting at relative location 400:

```
CALL WRITE ( file^a, array, 200 );
```

This single call to the file system actually involves two separate disc operations to two disc sectors.

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

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

Figure 4-4 on the following page shows a single WRITE that crosses sector boundries.

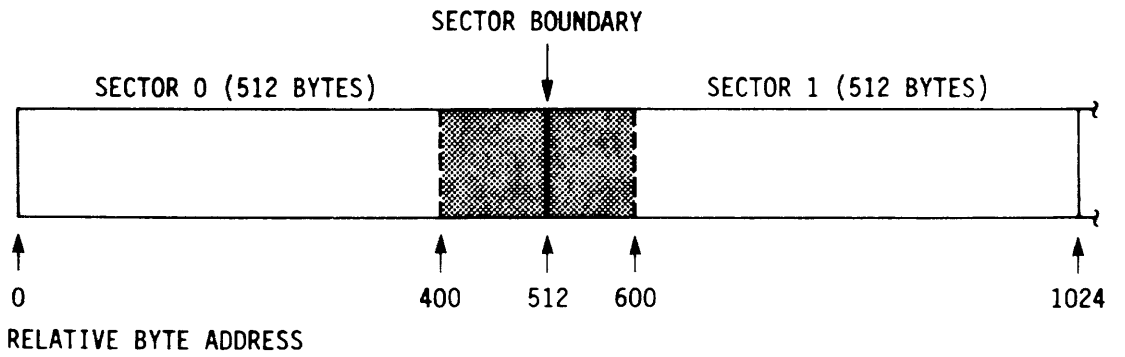


Figure 4-4. Example of Crossing Sector Boundries

Note that although full sector transfers are most efficient for the file system to perform, they are not necessarily the most efficient for a particular application. The application program can block data in its own memory area to accumulate a record of 512 bytes.

Resident Buffering (NonStop system only)

For unstructured files on a Tandem NonStop System, resident buffering is available. By using resident buffering, the data transferred because of an i/o request is transferred directly between the application process's data area and an i/o buffer in the processor module where the primary i/o process controlling a device is located. This bypasses the normal intermediate transfer to a file system buffer in the processor module where the application process is running. In addition to saving a move operation, using resident buffering also means that an application process will not be suspended, waiting for file system buffer space to become available when performing an i/o operation. The affect of using resident buffering is shown in Figure 4-5 below.

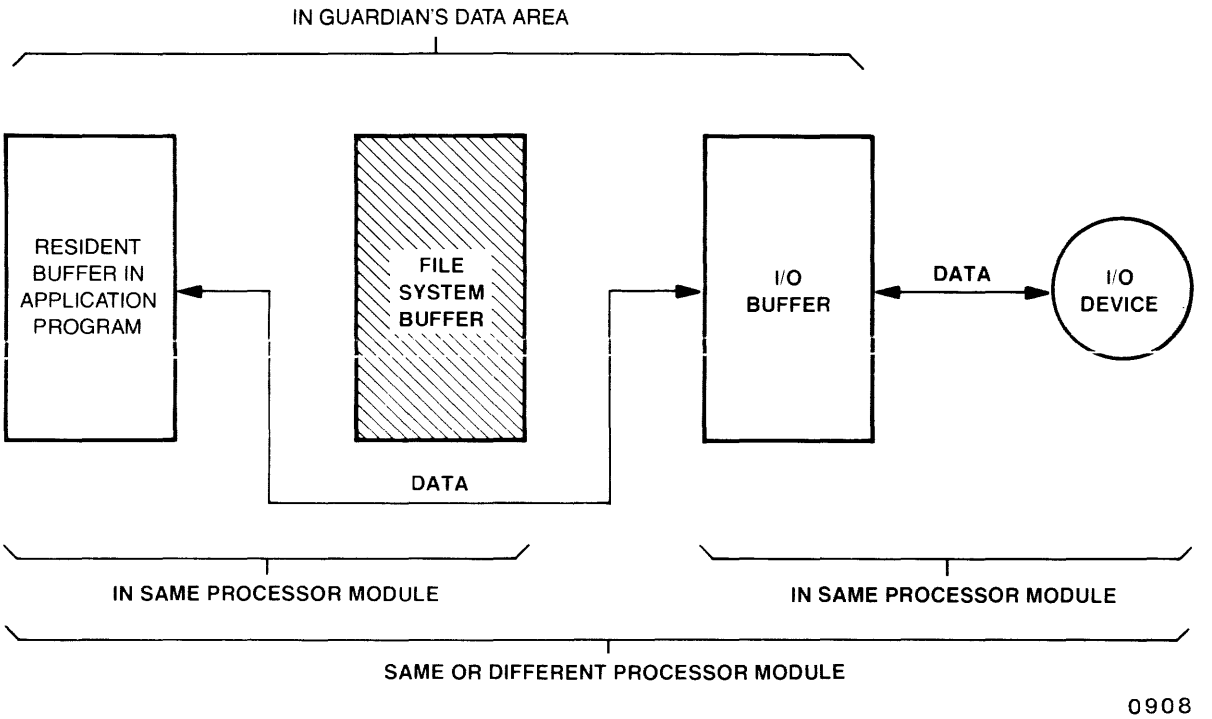


Figure 4-5. Resident Buffering

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

For example:

```
INT .buffer[0:255]; ! application buffer to be locked into memory.

INT PROC priv^lockdata ( address , count , sysmap ) CALLABLE;
   INT address, count, sysmap;
  BEGIN
    RETURN LOCKDATA ( address , count , sysmap );
  END; ! priv^lockdata.
```

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

LOCKDATA is invoked as follows:

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

specifies that the physical page(s) where "buffer" is located are to be made main-memory resident and are to be assigned to entries in the system data map. A "1" is returned in "n" if the page(s) are successfully locked.

Then OPEN is called, specifying resident buffers:

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

A subsequent call to a file system read or write procedure would then specify "buffer" in the procedure's <buffer> parameter. For example:

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

Note: For TMF, the buffer must be declared as .buffer [-12,<size>], and the LOCKDATA parameters changed accordingly.

When using resident buffering, the following considerations apply:

- Although resident buffering is specified on a file-by-file basis, the same resident buffer may be used for several different files (if, of course, the structure of the program permits).
- It is not necessary to call LOCKDATA before OPEN is called. However, LOCKDATA must be called before the first i/o transfer (i.e., READ, WRITE, CONTROL, etc.) with a file is performed.
- The resident buffer is not used for accesses to structured ENSCRIBE files; it is only used for unstructured files.

CONSIDERATIONS FOR BOTH STRUCTURED AND UNSTRUCTURED FILES

This section describes ENSCRIBE features that apply to both structured and unstructured files.

Locking -- General Concept

Access to a shareable file among two or more processes is coordinated through the use of file and record locking. A process requests a lock before performing a critical operation so that other accesses are temporarily excluded. A process performs an unlock when a critical operation is completed to allow other access by other processes.

File Locking

File locking is performed by calling the LOCKFILE and UNLOCKFILE procedures. If the file is unlocked when LOCKFILE is called for it, the file becomes locked and the caller to LOCKFILE continues executing. If the file is locked, then the action taken depends on the locking mode in effect at the time of the call (as described under File/Record Locking Interaction below). File unlocking is accomplished by calling the UNLOCKFILE procedure.

Record Locking

Individual records of a file are locked either by calling the LOCKREC procedure, which locks the record at the current position, or by calling the READLOCK or READUPDATELOCK procedures, which locks the record to be read before reading it. When a lock is requested for a record, if no other process has it locked, the lock is immediately granted. If the record or the file is locked, then the action taken depends on the locking mode in effect at the time of the call (as described under File/Record Locking Interaction below.) Record unlocking is accomplished by calling the UNLOCKREC procedure to unlock the current record, by calling the WRITEUPDATEUNLOCK procedure to unlock the record after it has been updated, or by calling UNLOCKFILE to unlock all records in the file locked by the caller. If a record is deleted, it is automatically unlocked. (If a record in a file audited by the Transaction Monitoring Facility, TMF, is deleted, the lock is not automatically relinquished. See the TMF Users Guide for details.)

Record locking has the advantage of allowing the maximum concurrency of access to a file while still guaranteeing the integrity of the file's contents when it is to be simultaneously updated by more than one process. However, for complex updates to a file which involve many records, record locking may not be desirable because of the amount of system processing required or because it increases the

possibility of "deadlock" (see "Deadlock", below). In this case, file locking can be used.

Locking Modes

Locks are granted on an open file (i.e., <file number>) basis. Therefore, if a process has multiple opens of the same file, a lock through one <file number> excludes accesses to the file through other <file numbers>.

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

- Default Mode

  With this mode, the requestor of a lock or read of a locked record (that is not locked by the <file number> supplied in the call), is suspended until the file/record becomes unlocked.

- Alternate Mode

  With this mode, a lock or read request of a locked record (that is not locked by the <file number> supplied in the call), is immediately rejected with a "file/record is locked" error indication (<error> = 73) This allows the requesting process to take alternative action.

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

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

File/Record Locking Interaction

The following description applies only if the default locking mode is in effect.

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

Similarly, for a record having more one or more pending lock requests, there is a queue of record lock requests. When a request is made to read from a locked record, but the record is not locked through the <file number> supplied in the call, the read request is queued with the record lock requests. When the current lock is cleared, the request at the head of the locking queue for the record is granted. If the request is a lock request, the lock is granted and the request continues processing; if the request is a read request, the request is completed.

A file lock is equivalent to locking all records in a file. If any records in a file are locked when a request is made for a file lock, the file lock is queued behind any record locks for the file. Conversely, if the file is locked when a request is made to lock a record, the record lock request is queued behind any file locks for the file.

There is an exception to the preceding statement; if a process has one or more records locked, then requests another record lock for that file, the record lock will preempt any pending file locks for that file (the request will not preempt other record locks for the same record). This exception minimizes the possibility of a "deadlock" condition occurring. This is illustrated as follows:

```
Process A                Process B                Process C

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

Note that a deadlock condition occurs if a process has a given file open more than once and a call to READ or READUPDATE is made by the process having the file locked but not locked via the <file number> supplied to READ or READUPDATE.

Deadlock

One problem that may occur when multiple processes require multiple record 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, the situation described above would never happen if both processes requested the lock to record 1 before they requested the lock to record 2.

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

Record Locking with Unstructured Files

So that applications can define their own file structures and still use the capabilities of record locking, record locking of unstructured files is permitted.

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

Unlocking is performed in the same manner as unlocking with structured files.

Record Locking Limitation

The maximum number of concurrent record locks that can be pending on a given file is 189. If this maximum is reached and an additional lock request is made, the lock request will be rejected with a "unable to obtain main memory space for a control block" error, <error> = 32.

Purge Data

An application process can programmatically purge all data from a file by means of the CONTROL procedure, "purge data" operation. This CONTROL operation does not physically purge data from a file. Rather data is "logically" purged by setting the file's current-record, next-record, and end-of-file pointers to relative byte 0. The end-of-file pointer in the file label on disc is also updated due to this operation.

For example, to purge all data from a file, the following call to CONTROL could be made:

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

sets the current-record, next-record, and end-of-file pointers to relative byte 0 and updates the end-of-file pointer copy in the file's file label on disc.

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

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

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

sets the end-of-file pointer to relative byte 0 then deallocates all extents.

Verify Write

Using verify write ensures the integrity of each write operation to a disc file. A byte-by-byte comparison of the just-written data on disc is made by the disc controller hardware with the corresponding data in memory. Note, however, that this requires an additional disc revolution.

Verify write is enabled by the SETMODE procedure (<function> = 3). The default setting disables verify write.

Refresh

While a file is open, the information in its FCB, such as the end-of-file pointer, is kept in main memory. To maximize performance, the end-of-file pointer is normally only written to the file's disc label when: a REFRESH procedure is executed for the file; a PUP REFRESH command is executed for the file; a CONTROL operation is executed for the file; an extent is allocated for the file; and, the last accessor closes the file. While refreshing the file's disc label only when these conditions occur maximizes system performance, the following considerations should be taken into account:

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

A CREATE auto-refresh option is available, indicated by setting <file type>.<10> of the CREATE procedure, that causes the File Label to be written to disc each time the end-of-file is advanced. The FUP ALTER (REFRESH) Command can also be used to specify the auto-refresh option (see the GUARDIAN Command Language and Utilities Manual, FUP). However, keeping this information in main-memory results in a significant increase in processing throughput.

For applications that cannot afford the overhead resulting from the auto-refresh create option, the File Label on disc can be forced to represent the actual state of a file through the use of the REFRESH procedure, or the equivalent Peripheral Utility Program (PUP) REFRESH command. The execution of the REFRESH procedure (or command) writes the information contained in any File Control Blocks (FCBs) to the File Labels on the associated disc volume.

REFRESH should be used in the following instances:

- Prior to a file backup in cases where the application is always running (i.e., files are open). At some point during the day when the system is quiescent (i.e. no transactions are taking place), a REFRESH command is issued for all volumes in the system. Then, when the files are backed up, the file labels on backup tape will represent the actual states of the files backed up.

- Prior to a total system shutdown. This will ensure that the file labels on disc will represent the states of files on disc.

- If it is deemed desirable to minimize the effect of a total system failure. Periodically (say, every ten minutes) an application process executes that calls the REFRESH procedure. Note that this is not necessary except in the instance when a power failure occurs that exceeds the limit of memory battery backup. Therefore, this action is unnecessary unless the computer site is susceptible to frequent and severe power outages.

Programmatic Extent Allocation

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

For example, to allocate all 16 extents in a newly created file, the file is opened then the following call to CONTROL is made:

```
LITERAL alloc^op = 21,
        max^ext  = 16;

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

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

Extent Allocation Errors

There are two errors associated with allocating disc extents: unable to obtain disc space for file extent (FILEINFO <error> = 43) and disc file full (FILEINFO <error> = 45).

An example showing both kinds of error -- a file is created with an extent size of 2,048 bytes, then repetitive writes of 400 bytes are executed to the file:

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

Writes one through five are successful ("num^written" = 400), write six fails after transferring 48 bytes of "buffer" to the disc ("num^written" = 48). If all disc space has been allocated to other files, the <error> returned by FILEINFO is 43 (out of disc space). If the current extent is the last one permitted in the file (i.e., extent number 15) then the <error> returned by FILEINFO is 45 (disc file full). This situation is illustrated in Figure 4-6 below.

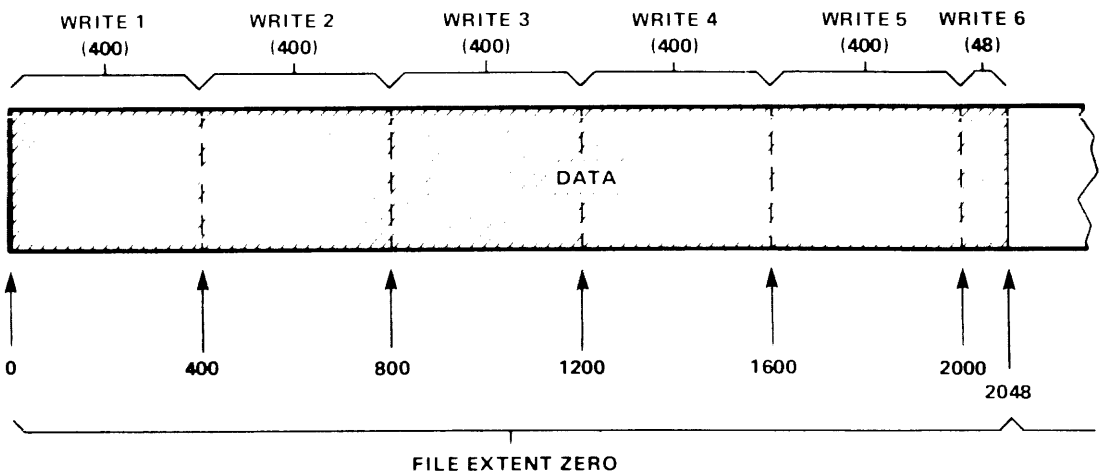


Figure 4-6. Example Showing Extent Allocation Error

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

Programmatic Extent Deallocation

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

For example, to deallocate any unused extents in a file, the file is opened then the following call to CONTROL is made:

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

   .
CALL CONTROL ( file^a, alloc^op, dealloc );
   .
```

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

SUMMARY OF DISC CONTROL AND SETMODE OPERATIONS

The following table gives a summary of the CONTROL and SETMODE operations that apply to disc files.

Table 4-2. Disc CONTROL and SETMODE Operations

```
Disc CONTROL Operations

 <operation>

 2 = write end-of-file (write access is required)

     <parameter> = none

20 = disc, purge data (write access is required)

     <parameter> = none

21 = disc, allocate/deallocate extents (write access is required)

     <parameter> = 0     = deallocate all extents past the end-of-
                           file extent

                   1:16 = number of extents to allocate for a
                          key-sequenced file

                   1:<total extents>
                        = total number of extents to allocate
                          for entry-sequenced, relative and
                          unstructured files
                    where

                      <total extents> = 16 * <number of partitions>


Disc SETMODE Functions

 <function>

 1 = set disc file security

 2 = set disc file owner id
```

→

Table 4-2. Disc CONTROL and SETMODE Operations (cont.)

Disc SETMODE Functions (cont.)

<function>

3 = set disc verify write

<parameter 1> = 0, means off (default setting)
= 1, means on

<parameter 2> is not used

4 = set disc lock mode

<parameter 1> = 0, default mode, process will be suspended when lock or read is attempted
= 1, alternate mode, lock or read attempt will be rejected with "file is locked" error (<error> = 73)

<parameter 2> is not used

57 = set disc serial or parallel writes (overrides SYSGEN setting for this file)

<parameter 1> = 1, serial writes
2, parallel writes

<parameter 2> is not used

ERRORS AND ERROR RECOVERY

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

The error number associated with an operation on an open file can be obtained by calling the FILEINFO procedure and passing the <file number> of the file in error:

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

The error number associated with an unopen file or a file open failure can be obtained by passing the <file number> as -1 to the FILEINFO procedure:

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

Note: the OPEN procedure returns -1 to <file number> if the open fails.

File Management Errors

File management errors are grouped into three major categories:

error	category
0	No error. Operation executed successfully.
1-9	Warning. Operation executed with exception of indicated condition. For <error> 6, data is returned in application process's buffer.
10-255	Error. Operation encountered an error. For data transfer operations, either none or part of specified data was transferred (with exception of data communication <error> 165, which indicates normal completion - data is returned in application process's buffer).
300-511	Error. These errors are reserved for process application-dependent use.

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

Errors occuring during disc file access can be separated into the following categories:

- Path Errors
- Data Errors
- Device Operation Error
- Failure of Primary Application Process

Path Errors

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

Data Errors

These are error numbers 50 through 59, 120 through 139, and 190 through 199. The file system automatically retries operations associated with this type of error. Therefore, if one of these errors is returned, all or part of the file can be considered invalid. For errors 50 through 59, the file may be recoverable through use of the COPY command with the RECOVER option of the Peripheral Utility Program (PUP). Error 120 (data parity error) causes the associated track to be flagged by the file system as bad. A bad track can be assigned to a spare track through use of the SPARE command of PUP.

Device Operation Error

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

Failure of Primary Application Process

This is not an disc error in the strictest sense. Rather, this is a failure of the processor module where the primary process of a primary/backup process pair is executing. Operations associated with this type of failure must be retried by the backup application process when it takes over the applications work. A discussion for recovering from this type of error is provided in section 5 "Checkpointing (Processor Module Failure Recovery)" of the GUARDIAN Programming Manual.

Errors Grouped by Error Number

This section groups of the file system errors by functional class and provides a description of each error class.

20-29 CODING ERROR
An invalid or out-of-bounds parameter was supplied to a file system call (errors 21, 22, 23, 29), or a call was made at an improper time (errors 25-28).

30-33 SYSTEM CONFIGURATION PROBLEM
A system configuration problem occurred in obtaining the desired space from a configurable memory space pool.

40 OPERATION TIMED OUT
This error can occur only with the AWAITIO procedure:

- If <file number> = -1 was specified (indicating "any file") and/or <awaitio timeout> = 0D (indicating a completion check) was specified, then the operation is considered incomplete (call AWAITIO again). Any indication other than <error> = 40 (i.e., CCE, CCG, or both CCL and <error> <> 40) indicates a completion.

- If a particular file was specified, operation did not complete within the specified <time limit>. In this case, <error> = 40 is considered a completion (i.e., the operation failed).

43 OUT OF DISC SPACE
Particular volume used is at its limit of available disc space (this may be due to fragmentation). Amount of available space and number and size of fragments on volume can be determined by using LISTFREE command of PUP (Peripheral Utility Program). Disc space can be recovered by (a) purging files, (b) closing temporary files, or (c) backing up all files on volume using BACKUP, then restoring the files to the volume using RESTORE.

50-58 DISC FILE INACCESSIBLE
Severe problem exists on disc volume. File associated with error is no longer accessible, although other files on same volume may be. It may be possible to recover files on volume by using COPY command of PUP with RECOVER option specified.

Errors 54-57 also imply that a disc hardware problem exists (file system retried the associated operation a number of times before reporting the error).

100-109 DEVICE REQUIRES ATTENTION
Human intervention may be necessary to return device to operable state.

110-112 TERMINAL ACCESS FAILURE
Error occurred with an application process using same terminal as another process or using operator console.

120-199 DEVICE HARDWARE PROBLEM
A device hardware problem exists. (Associated operation is retried several times before error is reported to application process).

Error 120, data parity error, occurring on a disc or tape READ or READUPDATE returns as much (invalid) data as possible. Number of characters returned can be determined by checking the procedure's <count read> parameter if a "wait" read or the <count transferred> parameter of AWAITIO if a "no-wait" read.

Error 121, overrun, if it occurs intermittently, also indicates that a physical configuration problem may exist.

Errors 160-178 occur only when using the ENVOY Data Communications Manager software. The errors are described in detail in the ENVOY Byte-Control Protocols Manual, P/N 82018.

Error 190, invalid status, indicates that a hardware problem exists with either the device or its controller.

200-255 PATH ERROR
Error occurred while attempting to use primary path to I/O device.

200-201: Operation never got started; medium movement did not take place.

210-231: Operation failed at some indeterminate point. If one of these errors has occurred, associated medium might have moved (it should be assumed so). Recovery procedure depends on device involved (see descriptions of individual device characteristics).

200,230, At least one path, and possibly 231: both paths, are operable.

201-229: At least one path is inoperable (i.e., automatic switch from primary path was made).

Special Considerations for Errors 200, 210, and 211

Error 200 indicates that a path switch occurred because of some other file access. This is a typical error when more than one process is accessing the same device or a process has more than one i/o operation outstanding to the same file at one time.

Error 210 indicates that a path switch to a hardware device controller occurred while this operation was in progress. This error is associated with concurrent operations involving more than one unit connected to a multi-unit controller. It occurs when an operation is in progress with one unit on a multi-unit controller and an error is detected during an operation with another unit on the same controller. (The other operation could have been on behalf of this or another application process.)

Error 211 indicates that the processor module controlling the device associated with this file operation has failed. The file operation itself has stopped at some indeterminate point.

The table in Appendix D lists all of the file management errors and gives a brief explanation of each error.

Error Recovery

The programmer must consider a number of items when writing error recovery routines:

- The type of device (e.g., disc, magnetic tape, line printer, etc.)
- The type of error (i.e., whether it is recoverable programmatically)
- The number of "no-wait" operations outstanding when the error was detected

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

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

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

- WRITE "no-wait" 1 initiated to printer print: "line one"
- WRITE "no-wait" 2 initiated to printer print: "line two"
- WRITE "no-wait" 3 initiated to printer print: "line three"

Then the three writes are completed with calls to AWAITIO:

- AWAITIO 1 indicates "no-wait" 1 succeeded (line printed)
- AWAITIO 2 indicates "no-wait" 2 failed (line not printed)
- AWAITIO 3 indicates "no-wait" 3 succeeded (line printed)

The following was printed as a result of the three "no-wait" writes:

```
"line one"
"line three"
("line two" is missing)
```

The rule of thumb is: when order is important, don't permit concurrent operations on the same file.

Error Handling For Structured Files

The following file management errors are peculiar to ENSCRIBE structured files:

warning/error	description
3	failure to open or purge a partition. This warning occurs when opening a partitioned file, if a partition defined for the file cannot be opened. The open succeeds but an attempt to access the non-existent partition will return an error 72, "attempt to access unmounted partition". This warning also occurs when a partitioned file is purged if a defined partition does not exist.
4	failure to open an alternate key file. This warning occurs when opening a file having alternate keys, if an alternate key file defined for the file cannot be opened or if the alternate key file was created with a record length that is not consistent with the alternate key definition in the primary file. The open succeeds but an attempt to access the file via an alternate key referenced in the unopened file will result in an error 46, "invalid key specified".

CAUTION

When an insertion or update is made to a file having an unopened alternate key file, the insertion (i.e., call to WRITE) will complete successfully (i.e., <error> = 0). However, no insertion or update can be made to the unopened alternate key file.

5	failure to provide sequential buffering. This warning occurs when sequential buffering is requested but exclusion mode = "share" is specified or the length of the sequential buffer is not sufficient to contain a data block from the file. The open succeeds, but normal system buffering is used.
10	record already exists (file is open). For relative files, this error indicates that an attempt has been made to insert a record into an occupied position; for key-sequenced files, an attempt has been made to insert a record having a primary key field value duplicating that of an existing record in the file; for a file having alternate keys, an attempt has been made to insert a record having a key field duplicating that of an existing record and "key is unique" has been defined for the field.

11 record not in file (file is open). This error can occur only when the READUPDATE or WRITEUPDATE procedures are called. For relative files, this error indicates that no record exists at the designated record number; for key-sequenced files, no record exists having the designated primary key field value; for files having alternate keys, no record exists having the designated key field value.

21 invalid count specified. This error is returned for several reasons. At file creation, in particular, this error is returned for a key-sequenced file if the record length is greater then one-half the block length - 24, for other structured files if the record length is greater than block length - 24, and for files having alternate keys if an invalid alternate key length is specified. During file access, this error is returned if a key length value is specified that is not consistent with the definition of the key. In particular, an insertion of a zero length primary key or a partial alternate key was attempted or the length supplied to KEYPOSITION for the <compare length> value exceeds the length defined for the key field.

46 invalid key specified. This error is returned for several reasons, some of which are: At file creation, a key field extends past the record length; for partitioned files, the partitions are not defined in ascending order of partial key field values; for alternate key files, either unique keys of differing lengths or unique and non-unique keys are defined for the same file. During file access, a <key specifier> was specified to KEYPOSITION that is not defined for the file; an attempt was made to position via an alternate key field for which the open previously failed; an attempt was made to update a non-existent record.

47 key not consistent with file data. This error is returned if, for some reason (such as no disc space available), the alternate key file cannot be updated on an insert, update, or delete operation to the primary file

72 attempt to access unmounted partition. This error is return if the volume on which the partition resides is not mounted or the partition does not exist (see warning 3).

Error Considerations for Key-Sequenced Files

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

Error Considerations for Files having Alternate Keys

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

Error Considerations For Partitioned Files

Each partition of a partitioned file is liable to get errors apart from other partitions of the file. This is especially significant for errors in the range of error numbers 42-45 as these are errors regarding disc space allocation.

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

For errors in the range of 42-45, it may be possible to alter the size characteristics of the partition where the error occurred by using the FUP program.

ACTION OF CURRENT KEY, KEY SPECIFIER, AND KEY LENGTH

The following table is intended as a concise definition of file management operations and their relationships to file currency information.

Table 4-3. Action of Current Key, Key Specifier, and Key Length

```
Definitions:
  CKV      = <current key value>
  CKS      = <current key specifier>
  CKL      = <current key length>
  CMPL     = <compare length>
  MODE     = <positioning mode>: approx   = 0
                                 generic  = 1
                                 exact    = 2
  primary  = 0
  next     = flag, if true, means that the next record in
             sequence is to be referenced
  rip      = relative file insertion pointer
  present  = true if parameter is supplied
  keyseq   = filetype = 3
  entryseq = filetype = 2
  relative = filetype = 1

  keyfield ( record, specifier );

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

  keylength ( record, specifier );

  ! returns the length of the "specified" key field in the record.
  ! if record = 0 then returns the defined key field length.

  find (mode, specifier, key value, compare length)

  ! returns the position of the first record in the file according
  ! to mode, specifier, key value, and compare length.
  !
  ! if mode = approx, positioning occurs to the first record whose
  ! key field, as designated by the <key specifier>, contains a
  ! value equal to or greater than <key>.  If no such record
  ! exists, an end-of-file indication is returned (<error> = 1)
  !
  ! if mode = generic, positioning occurs to the first record
  ! whose key field, as designated by the <key specifier>,
  ! contains a value equal to <key> for <compare length> bytes.
  ! If no such record exists, an end-of-file indication is
  ! returned (<error> = 1)
```

→

```
   ! if mode = exact, positioning occurs to the first record whose
   ! key field, as designated by the <key specifier>, contains a
   ! value of exactly <compare length> bytes and is equal to <key>.
   ! If no such record exists, an end-of-file indication is
   ! returned (<error> = 1)

   find^next (mode, specifier, key value, compare length)

   ! returns the position of the next record in the file according
   ! to mode, specifier, key value, and compare length.
   !
   ! if mode is approximate, positioning occurs to the next record
   ! in the file
   !
   ! if mode is generic, positioning occurs to the next record.  If
   ! the key field, as designated by the <key specifier>, is not
   ! equal to <key> for <compare length> bytes, an end-of-file
   ! indication is returned
   !
   ! if mode is exact, an end-of-file indication is returned.

   insert(key value, key length);

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

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

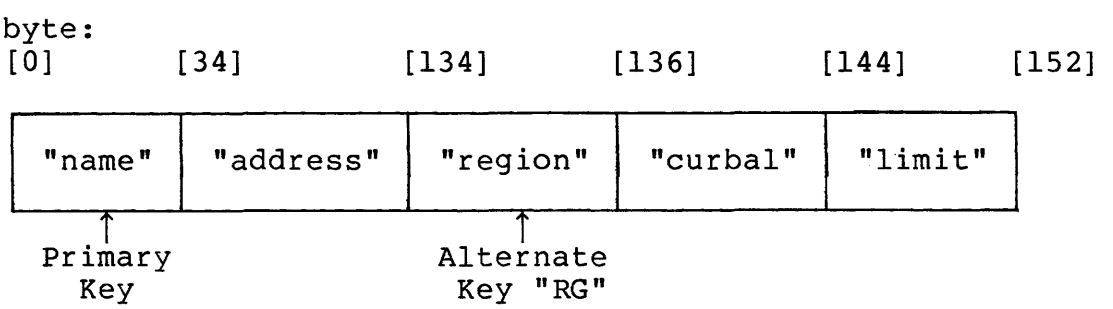
→

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


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

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

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

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

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

ACCESS EXAMPLES

Using the following "CUSTOMER" Record:




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

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

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

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

Contents of the "CUSTOMER" File:

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

Example 1. Action of Current Position

- Position via primary key "ROGERS"

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

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

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

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

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

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

Example 2. Approximate Subset by Primary Key Following Open

```
INT .cust^file^name[0:11],
      .
     cust^fnum,
       .
   .
CALL OPEN (cust^file^name, cust^fnum,..);
   .

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

Primary Key
|
v

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

Example 3. Approximate Subset by Alternate Key

Key specifier = "RG".

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

Alternate Key
"RG"
|
v

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

Example 4. Generic Subset by Primary Key

Primary key value = "BROWN".

```
key ':=' "BROWN";
compare^len := 5;
CALL KEYPOSITION ( cust^fnum, key, ,compare^len ,generic );

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

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

Example 5. Exact Subset By Primary Key

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

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

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

    .
    .
```

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

2   EOF
```

Example 6. Exact Subset by Non-Unique Alternate Key

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

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

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

Example 7. Insert of a Record to a Key-Sequenced File

Record to be inserted =

HEATHCLIFF	PORTLAND, OR.	WE	0000.00	0500.00

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

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

"CUSTOMER" File After Insert

"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	<-inserted
JONES	DALLAS, TX.	SO	1234.56	2000.00	
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	
ROGERS	BOISE, ID.	WE	1024.00	1500.00	
SANFORD	L.A., CA.	WE	0301.00	1000.00	
SMITH	DAYTON, OH.	NO	0010.00	0500.00	

Example 8. Random Update

EVANS	BUTTE, MT.	WE	0010.00	0100.00
HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00
JONES	DALLAS, TX.	SO	1234.56	2000.00


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

key ':=' "HARTLEY";
CALL KEYPOSITION (cust^fnum, key);
IF <> THEN ...
CALL READUPDATE (cust^fnum, cust, cust^len);
IF <> THEN ...
```

HARTLEY	CHICAGO, IL.	NO	0433.29	0500.00

```
  .
cust^curbal := cust^curbal + 30.00F
  .
CALL WRITEUPDATE (cust^fnum, cust, cust^len);
IF <> THEN ...
```

HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00

Example 9. Random Update to Non-Existent Record

Record to be updated = "BROWN, C"

ADAMS	MIAMI, FL.	SO	0000.00	0500.00
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
BROWN,B	BOSTON, MA.	EA	0301.00	1000.00
EVANS	BUTTE, MT.	WE	0010.00	0100.00

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

key ':=' "BROWN,C";
CALL KEYPOSITION (cust^fnum, key);
IF <> THEN ...
CALL READUPDATE (cust^fnum, cust, cust^len);
IF < THEN
  BEGIN
    CALL FILEINFO (cust^fum, error);
    IF error = 11 THEN .. ! record not found.
      .
      .
```

Example 10. Sequential Reading via Primary Key with Updating

The "limit" for each record having a "limit" >= 1000.00 and <= 2000.00 is raised to 2000.00.

```
compare^len := 0;
CALL KEYPOSITION ( cust^fnum, key, , compare^len);
                        ! position to first record via primary key.
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ ( cust^fnum, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
          IF cust^limit >= 1000.00F AND cust^limit <= 2000.00F THEN
            BEGIN
              cust^limit := 2000.00F;
              CALL WRITEUPDATE ( cust^fnum, cust, cust^len );
              IF < THEN ... ! error.
                .
                .
            END;
        END;
  END;  ! read loop.
   .
   .
```

				"limit"	
ADAMS	MIAMI, FL.	SO	0000.00	0500.00	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00	
BROWN,B	BOSTON, MA.	EA	0301.00	2000.00	<- updated
EVANS	BUTTE, MT.	WE	0010.00	0100.00	
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00	
JONES	DALLAS, TX.	SO	1234.56	2000.00	
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00	
RICHARDS	MINNI, MN.	NO	0000.00	0500.00	
ROGERS	BOISE, ID.	WE	1024.00	2000.00	<-updated
SANFORD	L.A., CA.	WE	0301.00	2000.00	<-updated
SMITH	DAYTON, OH.	NO	0010.00	0500.00	

Example 11. Random Deletion via Primary Key

Primary key = "EVANS"

BROWN,B	BOSTON, MA.	EA	0301.00	2000.00
EVANS	BUTTE, MT.	WE	0010.00	0100.00
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00


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

key ':=' "EVANS"
CALL KEYPOSITION (cust^fnum, key);
IF <> THEN ...
CALL WRITEUPDATE (cust^fnum, cust, 0);
IF <> THEN ...
```

BROWN,B	BOSTON, MA.	EA	0301.00	2000.00
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00

Example 12. Sequential Reading via Primary Key with Deletions

Each record having a "curbal" value of 0.00 is deleted.

```
CALL KEYPOSITION ( cust^fnum, key, , 0); ! position to first
                                         ! record via primary
                                         ! key.
cust^eof := 0;
WHILE NOT cust^eof DO
  BEGIN ! read loop.
    CALL READ ( cust^fnum, cust, cust^len);
    IF > THEN cust^eof := 1
    ELSE
      IF < THEN ... ! error.
      ELSE
        BEGIN ! process the record.
          IF cust^curbal = 0.00F THEN
            BEGIN
              CALL WRITEUPDATE ( cust^fnum, cust, 0 );
              IF < THEN ... ! error.
                .
                .
            END;
        END;
  END;  ! read loop.
   .
   .
```

			"curbal"	
BROWN,A	REEDLEY, CA.	WE	0256.95	0300.00
BROWN,B	BOSTON, MA.	EA	0301.00	2000.00
HARTLEY	CHICAGO, IL.	NO	0463.29	0500.00
JONES	DALLAS, TX.	SO	1234.56	2000.00
KOTTER	NEW YORK, NY.	EA	0089.00	0500.00
ROGERS	BOISE, ID.	WE	1024.00	2000.00
SANFORD	L.A., CA.	WE	0301.00	2000.00
SMITH	DAYTON, OH.	NO	0010.00	0500.00

Example 13. Positioning with a Relative or Entry-Sequenced File

For the following declarations

```
INT(32)
      rec^addr;

STRING
      primary^key = rec^addr;
```

positioning by primary key is done with:

```
CALL POSITION (fnum, rec^addr);
```

positioning to end-of-file is done with:

```
rec^addr := -1D;

CALL POSITION (fnum, -1D);
```

and the current primary key value (current position) can be obtained with either:

```
CALL FILEINFO (fnum,,,,,,,,,,,rec^addr);

   or

CALL FILERECINFO (fnum,,,,primary^key);
```

Example 14. Sequential Read of a Relative or Entry-Sequenced File

The read begins at the beginning of the file.

```
  .
CALL OPEN (file^name, fnum,..);
  .

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

Note that the preceding statements are functionally identical to the example for sequential access of a key-sequenced file via its primary key.

Example 15. Insert to a Specific Position in a Relative File

```
CALL POSITION (fnum, 12345D);
CALL WRITE (fnum, buffer, count);
IF < THEN
  BEGIN
    CALL FILEINFO (fnum, error);
    IF error = 10 THEN ... ! record already exists at 12345D.
      .
      .
  END;
```

Example 16. Append to the end of a Relative File

```
CALL POSITION (fnum, -1D);
WHILE 1 DO
  BEGIN
    .
    buffer ':=' data FOR (count + 1)/2;

        prepare a record to be written.
    .
    CALL WRITE (fnum, buffer, count);
    IF <> THEN   ... ! error.
    .
    CALL FILERECINFO (fnum,,,,primary^key);

       returns the <record number> of where the new record is
       appended.

    .
  END;
```

Example 17. Insert to Empty Positions in a Relative File

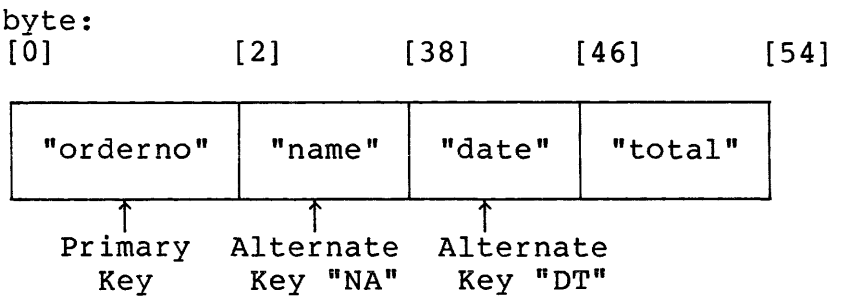
```
CALL POSITION (fnum, -2D);
WHILE 1 DO
  BEGIN
    .
    buffer ':=' data FOR (count + 1)/2;
    .
    CALL WRITE (fnum, buffer, count);
    IF <> THEN   ... ! error.
    .
    CALL FILERECINFO (fnum,,,,primary^key);

       returns the <record number> of where the new record is
       appended.

    .
  END;
```

Example 18. Append Records to an Entry-Sequenced File

```
WHILE 1 DO
  BEGIN
    .
    buffer ':=' data FOR (count + 1)/2;
    .
    CALL WRITE (fnum, buffer, count);
    IF <> THEN   ... ! error.

    .
  END;
```

RELATIONAL PROCESSING EXAMPLE

An "ORDER" Record:

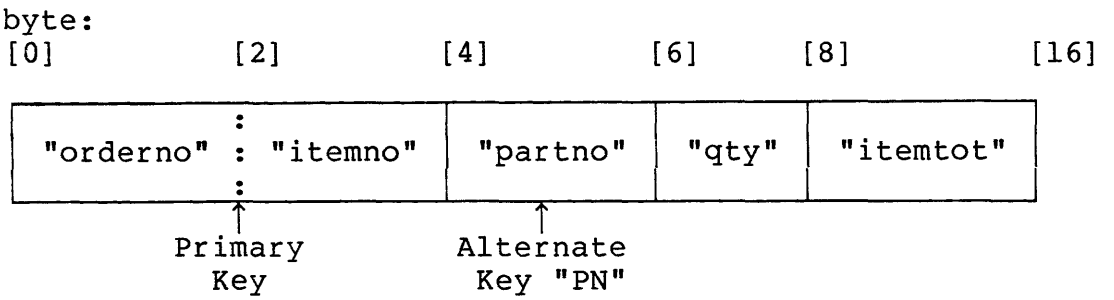



```
INT .order[0:26];                          ! order record.

STRING
    .sorder := @order '<<' 1;              ! byte addressable.

INT .order^orderno  = order;               ! order number field.

DEFINE
     order^len          =  54#,            ! order record length.
     order^name         =  sorder[2]#,     ! name field.
     order^name^len     =  36#,            ! name field length.
     order^date         =  sorderhrd[38]#, ! date field.
     order^date^len     =  8#;             ! date field length.

FIXED(2)
    .order^total        := @order[23];     ! total field.
                                           ! "total" = 0 means
                                           ! order not filled.
                                           ! "total" <> 0 means
                                           ! order filled but not
                                           ! shipped.
```

Contents of the "ORDER" File:

"orderno"	"name"	"date"	"total"
0020	SMITH	76/09/30	0000.00
0021	JONES	76/10/01	0000.00
0176	BROWN,B	76/10/17	0000.00
0180	ADAMS	76/10/17	0000.00
0410	SANFORD	76/10/22	0000.00
0498	ROGERS	76/11/02	0000.00
0568	EVANS	76/11/05	0000.00
0601	SMITH	76/11/08	0000.00
0621	RICHARDS	76/11/12	0000.00
0622	HARTLEY	76/11/12	0000.00
0623	KOTTER	76/11/12	0000.00

An "ORDER DETAIL" Record:




```
INT .orderdet[0:7];                          ! order detail record.

INT(32)
    .orderdet^orditem    := @orderdet;       ! order-item field.

DEFINE
     orderdet^len        =  16#,             ! order record length.
     orderdet^orderno    =  orderdet#,       ! order number subfield.
     orderdet^itemno     =  orderdet[1]#,    ! item number subfield.
     orderdet^partno     =  orderdet[2]#,    ! part number field.
     orderdet^qty        =  orderdet[3]#;    ! quantity field.

FIXED(2)
    .orderdet^itemtot    := @orderdet[4];    ! item total field.
                                             ! total = 0 means item
                                             ! not available.
```

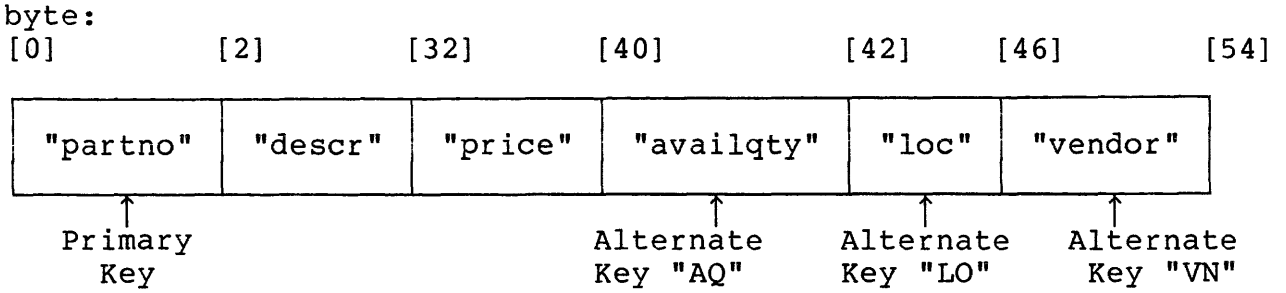
Contents of the "ORDER DETAIL" File:

"orderno":"itemno" "partno" "qty" "itemtot"

0020 : 0001	23167	00002	0000.00
0020 : 0002	02010	00001	0000.00
0020 : 0003	12950	00005	0000.00
0021 : 0001	00512	00022	0000.00
0021 : 0002	23167	00001	0000.00
0176 : 0001	32767	00001	0000.00
0180 : 0001	12950	00005	0000.00
0180 : 0002	32767	00022	0000.00
0180 : 0003	23167	00002	0000.00
0410 : 0001	01234	00010	0000.00
0410 : 0002	03379	00010	0000.00

0623 : 0012	01234	00010	0000.00

An "INVENTORY" Record:




```
INT .inv[0:27];                          ! inventory record.

STRING
    .sinv := @inv '<<' 1;                ! byte addressable.

DEFINE
     inv^len              = 54#,         ! inventory record length.
     inv^partno           = inv#,        ! part number field.
     inv^descr            = sinv[2]#,    ! part description field.
     inv^descr^len        = 30#,         ! description field length.
     inv^availqty         = inv[20]#,    ! avail. quantity field.
     inv^location         = sinv[42]#,   ! location field.
     inv^location^len     = 4#,          ! location field length.
     inv^vendor           = sinv[46]#,   ! vendor field.
     inv^vendor^len       = 8#;          ! vendor field length.

FIXED(2)
    .inv^price := @inv[16];              ! price field.
```

Contents of "INVENTORY" file:

"partno" "descr" "price" "availqty" "loc" "vendor"

00002	HI-FI	0129.95	00050	A01	TAYLOR
00512	RADIO	0010.98	00022	G10	GRAND
00987	TV SET	0200.00	00122	A76	TAYLOR
02010	TOASTER	0022.50	00000	F22	ACME
03379	CLOCK	0011.75	00512	A32	ZARF
12950	TOASTER	0020.45	00010	C98	SMYTHE
20211	WASHER	0314.29	00005	B44	SOAPY
23167	ANTENNA	0022.50	00008	A01	TAYLOR
32767	IRON	0025.95	00051	A82	HOT
65535	DRYER	0299.50	00022	Z02	SOAPY

The following example finds orders greater then one month old and fills them:

This involves the following steps

1. Reading the "order" file sequentially via the date field.

2. When an order is found that must be filled, the corresponding "customer" record is read (random processing) and an order header consisting of customer name and address is printed.

3 Next, the generic subset in the "order detail" file corresponding to the current "orderno" is read sequentially.

4. For each line item (i.e., record in the generic subset), the "inventory" file is read and updated (random processing), the line item record is updated, and the line item is printed.

5. When all line items for the current order have been processed, the order record is updated with the total price of the order. Then the customer current balance is updated and the total are printed.

The example code is

```
compare^len := 0;
! position to beginning of file via date field.
CALL KEYPOSITION (order^fnum,key,"DT",compare^len);
order^eof := 0;
WHILE NOT order^eof DO
  BEGIN ! reading order file via date field.
    CALL READ (order^fum, order, order^len);
    IF > OR order^date >= limit^date THEN order^eof := 1
    ELSE
      BEGIN ! fill order.
        ! read customer file.
        CALL KEYPOSITION (cust^fnum, order^name);
        CALL READUPDATE (cust^fnum, cust, cust^len);
        PRINT (order header);
        ! read order detail file for current order.
        compare^len := 2;
        CALL KEYPOSITION (orderdet^fnum,order^orderno,,
                          compare^len, generic);
        orderdet^eof := 0;
        WHILE NOT orderdet^eof DO
          BEGIN
            ! read line item.
            CALL READ (orderdet^fnum, orderdet, orderdet^len);
            IF > THEN orderdet^eof := 1
            ELSE
              BEGIN
                CALL KEYPOSITION (inv^fnum, orderdet^partno);
                CALL READUPDATE (inv^fnum, inv, inv^len);
                ! update inventory record
                CALL WRITEUPDATE (inv^fnum, inv, inv^len);
                ! update order detail record
                CALL WRITEUPDATE (orderdet^fnum, orderdet,
                                  ordetlen);
                ! print the line item
                PRINT (line item)
              END;
          END;
        ! update the order file
        CALL WRITEUPDATE (order^fnum, order, order^len);
        ! update the customer file.
        CALL WRITEUPDATE (cust^fnum, cust, cust^len);
        PRINT (total);
      END; ! of fill order.
  END; ! of read order file via date field.
```

Records and files used to fill the first order:

From the "ORDER" file -

0020	SMITH	76/09/30	0000.00

From the "CUSTOMER" file -

SMITH	DAYTON, OH.	NO	0010.00	0500.00

From the "ORDER DETAIL" file -

0020 : 0001	23167	00002	0000.00
0020 : 0002	02010	00001	0000.00
0020 : 0003	12950	00005	0000.00

From the "INVENTORY" file -

23167	ANTENNA	0022.50	00008	A01	TAYLOR
02010	TOASTER	0022.50	00000	F22	ACME
12950	TOASTER	0020.45	00010	C98	SMYTHE

Records and files after filling the first order:

From the "ORDER" file -

0020	SMITH	76/09/30	0147.25

From the "CUSTOMER" file -

SMITH	DAYTON, OH.	NO	0157.25	0500.00

From the "ORDER DETAIL" file -

0020 : 0001	23167	00000	0045.00	
0020 : 0002	02010	00001	0000.00	<- not filled
0020 : 0003	12950	00000	0102.25	

From the "INVENTORY" file -

23167	ANTENNA	0022.50	00006	A01	TAYLOR	
02010	TOASTER	0022.50	00000	F22	ACME	<- none
12950	TOASTER	0020.45	00005	C98	SMYTHE	

SECTION 5

ENSCRIBE FILE CREATION

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

Characteristics of ENSCRIBE file creation:

- An ENSCRIBE file is created through use of the Tandem-supplied FUP program or through a programmatic call to the CREATE procedure. The DDL compiler will optionally generate a FUP command file that can be used to create files described by the DDL data base source schema.
- All partitions of a multi-volume file are automatically created when the first partition is created.
- For each structured file having one or more alternate key fields, the user must create the associated alternate key file(s). An alternate key file is created as a key-sequenced file.

The following topics must be considered when creating an ENSCRIBE disc file:

- File Type
- File Code
- Extents
- Logical Records
- Blocks
- Characteristics of Key-Sequenced Files
- Characteristics of Structured Files having Alternate Keys
- Characteristics of Partitioned (Multi-Volume) Files

CONSIDERATIONS FOR BOTH STRUCTURED AND UNSTRUCTURED FILES

This section describes file creation parameters common to all file types.

File Type

There are four file types:

- Key-Sequenced
- Relative
- Entry-Sequenced
- Unstructured

Key-Sequenced Files

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

Relative Files

A relative file consists of a set of fixed length records. Each record is uniquely identified among other records in a relative file by a record number; a record number denotes an ordinal position in a file. The first record in a relative file is designated by record number zero; succeeding records are designated by ascending record numbers in increments of one. A record occupies a position in a file whether or not the position has been written in.

Entry-Sequenced Files

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

Unstructured Files

An unstructured disc file is essentially a byte array, starting at byte zero and continuing to the last byte of the file, marked by the end-of-file pointer. The file system imposes no stucture on an unstructured disc file. How data is grouped into records and where records are located within the file is the responsibility of the application process. Unstructured files are accessed on the basis of a relative byte address. The current location in a file, the current rba, is automatically incremented by the file system following a READ or a WRITE. The application process can position to a specific location within the file by supplying the file system an rba with the POSITION procedure. Following a call to POSITION, the application process can issue any other file management call (READ, WRITE, READUPDATE, etc.).

Extents

Physical storage for a disc file is allocated by the file system in the form of discontiguous file extents. A file extent itself is a contiguous block of storage, starting on a sector boundary, containing a multiple of 2,048 bytes. The file system permits a maximum of 16 file extents to be allocated to a file.

The first extent of the file is called the "primary extent." Its size may be different from the size of extents 1-15, which are called "secondary extents." This allows a file to be created having a large primary extent to contain all the data to be initially placed in the file, and smaller secondary extents so that as the file grows, a minimum amount of disc space is allocated.

File Code

An application-defined file code can be assigned to a disc file at creation time. The file code is typically used to categorize files by the information they contain. File codes 100 through 999 are reserved for use by Tandem Computers, Inc.

Partitioned (Multi-Volume) Files

A file may be partitioned into up to 16 disjoint sub-files each of which resides on a different volume. Moreover, each volume can be on a different node in a network.

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

Partitioned files can be of value for a number of reasons. The most obvious one is that a file may be created whose size is not limited by the size of a physical disc pack. In addition, by spreading a file across more that one volume, the concurrency of access to records in the file can be increased. If the file is located on multiple volumes on the same controller, the operating system will take advantage of the controller's overlapped seek capability (i.e. many drives may 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 bandwidth). And if the file resides on volumes connected to controllers on different processors, the system will perform overlapped processing of requests and overlapped transfers not limited by the bandwidth of a single i/o channel.

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

EXTENTS FOR PARTITIONED FILES. Each partition may have a different primary and secondary extent size.

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

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

CONSIDERATIONS FOR STRUCTURED FILES

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

Logical Records

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

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

- Records in a file can be of varying lengths but cannot be larger than the <record length> defined at file creation.
- The maximum possible record size is determined by the specified data block size. For relative and entry-sequenced files, the maximum record length cannot exceed <data block size> - 24. This means that for relative and entry-sequenced files with a data block size of 4096 (the maximum), the maximum record length is 4072. For key-sequenced files, the maximum record length cannot exceed 1/2 * (<data block size> - 26). This means that for key-sequenced files with a data block size of 4096, the maximum record length is 2035.

Blocks

The block is the unit of information which is transferred between the file system and a disc volume. A block consists of one or more logical records and associated control information (the control information is used only by the system).

The following points should be considered when choosing a structured file's block length:

- Block length must be a multiple of sector size (512 bytes) and can not be greater than 4096 bytes.
- Regardless of the record length, the maximum number of records that can be stored in a single block is 511.

- Block length must include 22 bytes for block control and 2 bytes per record for record control. Therefore the number of records per block is

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

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

  Records may not span blocks. Therefore the block length must be at least <record length> + 2 + 22.

  For key-sequenced files, the data block size determines the maximum record length that can be defined for the file; the record length cannot exceed 1/2 * (<data block size> - 26).

Considerations for Key-Sequenced Files

Considerations for key-sequenced files include:

- Compression
- Primary Key
- Index Blocks

COMPRESSION. An optional "front compression" technique is provided so that more data can be stored in a given disc area, thereby reducing the number of head repositionings.

ENSCRIBE accomplishes front compression by eliminating leading characters that are duplicated from one record to the next. Instead, a count of duplicate characters is written in the first byte of the record.

For example, if following three records are inserted in a file

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

the following is actually written on the disc

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

The following should be considered when deciding if compression should be used:

- If compression is used, one additional byte per record may be required in each block. Moreover, there is additional system processing required to expand the compressed records.
- If data compression is used, the record's primary key field must begin at offset [0] of the record. Therefore, variable length primary keys cannot be used unless the entire record is the primary key field.
- If there is considerable similarity among the record's primary key values then data compression is desireable. If not, then compression just adds unnecessary system overhead.
- If there is enough similarity among records that the first records of successive blocks have similar primary key values, then index compression is desireable.
- Data compression is useful for alternate key files where several alternate keys tend to have the same value.

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

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

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

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

- If data compression is to be used, then the primary key field must be at the beginning of the record.
- If the primary key field is the trailing field in the record, the the primary key values can be of variable lengths.

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

INDEX BLOCKS. The length, in bytes, of a file's index blocks must be specified. The length must be a multiple of 512 and may not be greater than 4096.

Some considerations when choosing the index block length are

- If not specified, the index block length defaults to the data block length.
- Longer index blocks require more space in the Cache buffer. However, a longer index block may reduce the number of indexing levels and, therefore, accesses to the disc.

Considerations for Files Having Alternate Keys

Considerations for files having alternate keys include:

- Key Specifier
- Key Offset and Length
- Null Value
- No Automatic Update
- Alternate Key File(s)
- Unique Alternate Keys

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

KEY OFFSET AND LENGTH. The offset in the record where the alternate key field begins and the length of key field must be specified for each alternate key.

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

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

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

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

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

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

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

- Certain fields may not be referenced until a later date. Therefore, they can be updated in a "batch" (one pass) mode more efficiently.
- A field may have multiple "null" values. In this case, the application program must have the alternate key file open separately. The program must determine whether or not the field contains a "null" value. If it does not, the application program then inserts the alternate key reference into the alternate key file.

UNIQUE ALTERNATE KEYS. An alternate key field can be designated to contain a unique value. If an attempt is made to insert a record that duplicates an existing record's value in a unique key field, the insert operation is rejected. A "duplicate record" error indication will be returned to the application program in a subsequent call to FILEINFO. This is unlike non-unique alternate keys where the above operation would be permitted.

If a file has one or more unique alternate keys, the following must be considered:

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

ALTERNATE KEY FILE(S). For each primary structured file having one or more alternate keys, at least one corresponding alternate key file must be created by the user. Characteristics of alternate key files are:

- A single alternate key file can contain references to one or more alternate keys.
- A primary file can have a separate alternate key file for each group of one or more alternate keys. Some reasons to have separate alternate key files are:
  - The alternate key file can be smaller than if many alternate key references are in the same file. This results in less indices being referenced to locate a given alternate key.
  - An alternate key could be updated frequently, causing fragmentation of the file. The file could be "rebuilt" later.
  - The alternate key is unique.

  Some reasons not to have separate alternate key files are:
  - System control block space is allocated for each open of an alternate key file (i.e., open of the primary file).
  - A File Control Block is allocated for the first open of an alternate key file.
- The file type for an alternate key file is key-sequenced.
- The record length for an alternate key file is

```
   2  for the <key specifier>
 + the maximum <key length> of all alternate keys referenced
 + the <key length> of the associated primary key
```

  As alternate key files are key-sequenced files, the maximum record length is limited to 1/2 * (<data block size> - 26).
- The primary key length for an alternate key file containing non-unique key references is the same as its record length.
- The primary key length for an alternate key file containing unique key references is

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

- The <key offset> in all cases is zero.
- Alternate key files can be partitioned to span multiple volumes.

THE FILE UTILITY PROGRAM (FUP)

Typically, ENSCRIBE files are created through use of the File Utility Program (FUP).

The FUP commands related to file creation are:

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

For creating disc files, the FUP Program is intended to be used in the following manner:

- The user sets the creation parameters describing the file to be created by executing one or more SET commands.

  The SET command has a feature that allows the creation parameters to be set to match those of an existing file. This is useful for duplicating file structures.

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

- Once the creation parameters have the desired settings, the file is created by executing a CREATE command. An option to the CREATE command permits the user to override current creation parameter setting(s).

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

Running FUP

The File Utility Program resides in a file designated

$SYSTEM.SYSTEM.FUP

Normally, it is run through use of the Command Interpreter program. The command to run FUP is:

```
FUP [ / [ IN <command file> ] [, OUT <list file> ] / ]

      [ <command> ]
```

where

IN <command file>

specifies disc file, non-disc device, or process where FUP reads commands. FUP reads 132-byte records from <command file> until the end-of-file is encountered. Only one command is permitted per record. If <command file> is omitted, the home terminal is used.

OUT <list file>

specifies a non-disc device, process, or existing disc file where FUP directs its listing output (unless directed elsewhere by a command parameter). If the <list file> is an unstructured disc file, each <list file> record is 132 characters (partial lines are blank filled through column 132). If omitted, the home terminal is used.

<command>

is a FUP command, limited to 132 characters. If <command> is included, it is executed, then FUP terminates. For a complete description of all the FUP commands, see the GUARDIAN Command Language and Utilities Manual.

example

```
FUP INFO *
```

The FUP SET Command

The SET Command is used to set one or more creation parameter values for a subsequent creation. Parameter values can be specified explicitly or can be set to match those of an existing file.

The form of the SET Command is:

```
SET <create param> , ...

where

   <create param> is one of

      { LIKE <file name>                                              }
      {                                                               }
      { TYPE <file type> ! { U | R | E | K | 0 | 1 | 2 | 3 }           }
      { CODE <file code> ! {0:65535}                                  }
      { EXT { <extent size>                                         } }
      {     { ( <pri extent size> , <sec extent size> )             } }
      {                                                               }
      { REC <record length>          ! {1:4072}                       }
      { BLOCK <data block length>    ! {1:4096}                       }
      {                                                               }
      { IBLOCK <index block length> ! {1:4096}                        }
      { [ NO ] COMPRESS                                               }
      { [ NO ] DCOMPRESS                                              }
      { [ NO ] ICOMPRESS                                              }
      { KEYLEN <key length>          ! {1:255}                        }
      { KEYOFF <key offset>          ! {0:2034}                       }
      {                                                               }
      { ALTKEY ( <key specifier> { , <altkey param> } ... )           }
      {                                                               }
      {   <altkey param> is one of                                    }
      {                                                               }
      {     FILE <key file number>   ! {0:255}                        }
      {     KEYOFF <key offset>      ! {0:4071}                       }
      {     KEYLEN <key length>      ! {1:255}                        }
      {     NULL <null value>        ! { "<c>" | {0:255} }            }
      {     UNIQUE                                                    }
      {     NO UPDATE                                                 }
      {                                                               }
      { ALTFILE ( <key file number> , <file name> )                   }
      { [ NO ] ALTCREATE                                              }
      {                                                               }
```

→

```
{ PART ( <secondary partition num> , <volume name>                    }
{        [ , <pri extent size> [ , [<sec extent size>]                }
{        [ , <partial key value> ] ] ] )                              }
{                                                                     }
{ [ NO ] PARTONLY                                                     }
{ ODDUNSTR                                                            }
{ [ NO ] REFRESH                                                      }
{ [ NO ] AUDIT                                                        }
```

example

```
SET TYPE K, KEYLEN 36
```

Notes:

1. The current settings for <create params> can be displayed by means of the SHOW command.
2. The <create params> can be reset to their default settings by means of the RESET command.

Parameters for all File Types

The following five parameters apply to all file types:

- LIKE

  is used to set <create params> to the values of those of an existing file. A partial <file name> is expanded

  If <file name> specifies a secondary partition of a partitioned file, the PARTONLY <create param> will be set (see "PARTONLY", below).

- TYPE

  is used to set the file type. Legitimate values for <file type> and their meanings are:

  U or 0 = unstructured
  R or 1 = relative
  E or 2 = entry-sequenced
  K or 3 = key-sequenced

  The default setting for <file type> is "U".

- CODE

  is used to set the file code. <file code> is an integer in the range of {0:65535}. <file code> values 100-999 are reserved for use by Tandem Computers Inc. The default setting for <file code> is 0.

- EXT

  is used to set the extent size. Legitimate values for extent sizes and their meanings are:

```
{0:65535}              = number of pages (2048-byte units)
{0:65535} PAGES        = number of pages
{0:2099999999} BYTES   = number of bytes
{0:2099999999} RECS    = number of <record length> records
{0:65535} MEGABYTES    = number of million-byte units
```

  If the "RECS" form is used, the calculation will be based on the latest settings for <record length> (REC), <data block length> (BLOCK), <index block length> (IBLOCK), key field length and compression settings.

  The default setting for extent sizes is 1 page.

- [NO] REFRESH

  specifies that the file label is written to disc each time the end-of-file is modified. Either FUP or the file system CREATE call can set the REFRESH option. The setting defaults to no refreshing.

  Example for a new file creation:

```
FUP
SET LIKE <old-filename>
SET REFRESH
CREATE <new-filename>
EXIT
```

Parameters for Structured Files

The following two parameters apply to key-sequenced, relative, and entry-sequenced files:

- REC

  is used to set the record length. <record length> is an integer in the range of {0:4072} for relative and entry-sequence files, {0:2035} for key-sequence files, and {0:4096} for unstructured files. The default setting for <record length> is 80 bytes.

- BLOCK

  is used to set the data block length. <data block length> is an integer in the range of {0:4096}. The default setting for <data block length> is 1024 bytes.

Parameters for Key-Sequenced File Structures

The following six parameters only apply to key-sequenced files:

- IBLOCK

  is used to set the index block length. <index block length> is an integer in the range of {0:4096}. The default setting for <index block length> is 1024 bytes.

- [NO] COMPRESS

  is used to set/clear the states of both index and data compression. The default setting for this specification is NO COMPRESS.

- [NO] DCOMPRESS

  is used to set/clear the state of data compression. The default setting for this specification is NO DCOMPRESS.

- [NO] ICOMPRESS

  is used to set/clear the state of index compression. The default setting for this specification is NO ICOMPRESS.

- KEYLEN

  is used to set the primary key length. <key length> is an integer in the range of {1:255}. This specification must be made for key sequenced file structures, otherwise a creation attempt will fail.

- KEYOFF

  is used to set the primary key offset. <key offset> is an integer in the range of {0:4071}. The default setting for <key offset> is 0.

Parameters for Structured Files having Alternate Key Fields

The next three parameters only apply to key-sequenced, relative, and entry-sequenced files having alternate key fields:

- ALTKEY

  is used to set an alternate key specification. Each alternate key must be specified separately. The form of the ALTKEY specification is

  ALTKEY (<key specifier> { , <altkey param> } ...)

    <key specifier>

      is a two-byte value that uniquely identifies this alternate key field. This value is passed to the KEYPOSITION procedure when referencing this key field. <key specifier> is specified as either

        "[<c1>]<c2>"

          a one- or two-character string within quotation marks (if <c1> is omitted, then <c1> is treated as a zero (0)); or

        {-32768:65535}

          an integer that can be represented within 16 bits. A value of zero is not allowed.

<altkey param> is one of

```
FILE <key file number> ! {0:255}
KEYOFF <key offset>    ! {0:4071}
KEYLEN <key length>    ! {1:255}
NULL <null value>      ! { "<c>" | {0:255} }
UNIQUE
NO UPDATE
```

where

FILE

is used to set the key file number for <key specifier>. <key file number> is an integer in the range of {0:255}. <key file number> is related to an actual file by means of the ALTFILE <create param>. The default setting for <key file number> is 0.

KEYOFF

is used to set the key offset for <key specifier>. The default setting for <key offset> is 0.

KEYLEN

is used to set the key length for <key specifier>. This must be specified. Otherwise a subsequent creation attempt will fail.

NULL

is used to set a null value for <key specifier>. <null value> is an ascii character within quotation marks or an integer in the range of {0:255}. The default setting for this specification is "key does not have null value"

UNIQUE

is used to set "key is unique" for <key specifier>. The default setting if this is omitted is "key is not unique".

NO UPDATE

is used to set "no automatic updating" for <key specifier>. The default setting for this specification is "key is automatically updated".

- ALTFILE

  is used to set the file name of an alternate key file. This must be specified for each different <key file number> specified in an ALTKEY specification. The form of ALTFILE specification is

  ALTFILE (<key file number> , <file name>)

  <key file number>

  is an integer in the range of {0:255} specifying the alternate key file being named

  A partial <file name> is expanded using the default system, volume and subvolume names.

- [NO] ALTCREATE

  is used to set/clear automatic alternate key file creation. If ALTCREATE is specified, the alternate key file(s) will be created when the primary file is created. The default setting for this specification is ALTCREATE.

Parameters for Partitioned Files

The following two parameters apply to partitioned files only:

- PART

  is used to set secondary partition specifications for partitioned files. Each secondary partition must be specified separately. The form of the PART specification is

  PART (<secondary partition num> , <volume name>

  [, <pri extent size> [, [<sec extent size>]

  [, <partial key value>]]])

  <secondary partition num> , <volume name>

  specifies the volume where this secondary partition is to reside. <secondary partition num> designates the secondary partition; it is specified as an integer in the range of {1:15}; <volume name> specifies the volume (and possibly the network node). Note that the file name of file and the volume of the primary partition is specified when the file is created.

  <volume name> can be specified with either of the following two forms:

$<volume name>	(local form)
\<sys num><volume name>	(network form)

  <partial key value>

  for key-sequenced files only, specifies the lowest key value that will reside in this partition. <partial key value> is specified as

  "<c1><c2>...<cn>"

  a string of characters within quotation marks, or

  "[" { "<c>" | {0:255} } , ..."]"

  a list of single characters, each within quotation marks, and/or integers in the range of {0:255} separated by commas

  <partial key value> must be included for each partition of a key-sequenced file. (Note, for the primary partition the partial key value is zero (0).)

- [NO] PARTONLY

  specifies whether a subsequent creation is to create all partitions of a partitioned file ("NO PARTONLY") or a subsequent creation is to create a single partition ("PARTONLY"). The default setting, "NO PARTONLY", causes all partitions to be created.

  If PARTONLY is specified and a PART specification is in effect, the subsequently created file will be designated a primary partition. Conversely, if no PART specification is in effect, the subsequently created file will be designated a secondary partition. In either case, the <file name> of the (primary or secondary) partition is specified when the file is created.

Parameter for Files Audited by TMF

The following parameter applies to files within a system that has the Transaction Monitoring Facility (TMF).

- [NO] AUDIT

  is used to describe a file as either audited or not audited. See the Transaction Monitoring Facility (TMF) Users Guide for details.

Parameter for Unstructured Files

The following parameter applies to unstructured files:

- ODDUNSTR

  ENSCRIBE unstructured files exist in two ways. The even unstructured file rounds up any odd byte counts to the next even number, 3 goes to 4, or 5 to 6,... when reading, writing or positioning in the file. Unstructured files default to this setting. The odd unstructured file prevents rounding up, therefore reading, writing or positioning occurs at the byte count given. When odd unstructured is desired to prevent rounding up, the ODDUNSTR parameter must be specified.

  Example:

```
SET CODE 200, EXT 2048, ODDUNSTR
```

The FUP SHOW Command

The SHOW Command is used to display the current settings of the creation parameter values.

The form of the SHOW Command is:

```
SHOW [ / OUT <list file> / ] [ <create spec> , ... ]

where

   <create spec> is one of

      { TYPE                              }
      { CODE                              }
      { EXT                               }
      {                                   }
      { REC                               }
      { BLOCK                             }
      {                                   }
      { IBLOCK                            }
      { COMPRESS                          }
      { DCOMPRESS                         }
      { ICOMPRESS                         }
      { KEYLEN                            }
      { KEYOFF                            }
      {                                   }
      { REFRESH                           }
      { AUDIT                             }
      {                                   }
      { ALTKEY [ <key specifier> ]        }
      { ALTKEYS                           }
      { ALTFILE [ <key file number> ]     }
      { ALTFILES                          }
      { ALTCREATE                         }
      {                                   }
      { ODDUNSTR                          }
      {                                   }
      { PART [ <partition num> ]          }
      { PARTS                             }
      { PARTONLY                          }

   Each <create spec> keyword returns the current setting of the
   corresponding <create param>.  Omitting <create spec> returns
   all current settings applicable to the current file type
```

The FUP CREATE Command

The CREATE Command is used to create a file using the current creation parameter values (i.e., the default settings or those set via the SET command). The current parameter values can be overridden in the CREATE command by specifying alternate values for designated parameters.

The form of the CREATE Command is:

```
CREATE <file name> [ , <create param> ] ...

where

   <file name>

     is the name of the file to be created.  A partial file name
     is expanded

   <create param>,

     if included, overrides the corresponding current creation
     parameter setting for this creation.  See "SET Command" for
     <create param> format

example

   CREATE myfile
```

THE FUP RESET COMMAND

The RESET Command is used to reset one or more creation parameter values to their default settings.

The form of the RESET Command is:

```
RESET [ <create spec> , ... ]

where

   <create spec> is one of

      { TYPE                                }
      { CODE                                }
      { EXT                                 }
      {                                     }
      { REC                                 }
      { BLOCK                               }
      {                                     }
      { IBLOCK                              }
      { COMPRESS                            }
      { DCOMPRESS                           }
      { ICOMPRESS                           }
      { KEYLEN                              }
      { KEYOFF                              }
      {                                     }
      { ALTKEY [ <key specifier> ]          }
      { ALTKEYS                             }
      { ALTFILE [ <key file number> ]       }
      { ALTFILES                            }
      { ALTCREATE                           }
      {                                     }
      { ODDUNSTR <unstructured files>       }
      { [ NO ] REFRESH                      }
      { [ NO ] AUDIT                        }
      {                                     }
      { PART [ <partition num> ]            }
      { PARTS                               }
      { PARTONLY                            }

   Each <create spec> keyword resets the corresponding creation
   parameter to its default setting.  Omitting <create spec>
   resets all creation parameters to their default settings.
```

The FUP INFO Command

The INFO Command is used to display disc file characteristics.

The form of INFO Command is:

```
                                                    { DETAIL        }
                                                    { STAT[ISTICS]  }
INFO [ / OUT <list file> / ] <fileset list> [, { EXTENTS       } ]
```

where

<fileset list>

is a list of files whose characteristics are to be listed. <fileset list> is of the form

```
{ <fileset>           }
{ ( <fileset> , ... ) }
```

<fileset> is of the form

```
[\<system name>.][$<volume name>.][<subvol spec>.]<disc file spec>
```

\<system name>,

omitting \<system name> designates the home node

$<volume name>,

omitting $<volume name> designates the default volume

<subvol spec> is either

<subvol name> or "*"

"*" designates all subvols on the designated volume. If "*" is given, then <disc file spec> must also be given as "*". Omitting <subvol spec> designates the default subvolume

<disc file spec> is either

<disc file name> or "*"

"*" designates all files in the designated subvolume.

→

DETAIL

provides detailed information regarding file characteristics (see "DETAIL Option Listing Format" in the FUP section of the GUARDIAN Command Language and Utilities Manual for details.)

STAT[ISTICS]

provides information given by the DETAIL option plus summarized data concerning the usage of records and blocks in ENSCRIBE file structures (see "STATISTICS Option Listing Format" in the FUP section of the GUARDIAN Command Language and Utilities Manual for details.)

EXTENTS

provides a listing of extent allocation by file (see "EXTENTS" Option Listing Format in the FUP section of the GUARDIAN Command Language and Utilities Manual for details.)

example

```
INFO myfile,DETAIL
```

CREATE

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

The call to the CREATE procedure is:

```
CALL CREATE (    <file name>
             , [ <primary extent size>   ]
             , [ <file code>             ]
             , [ <secondary extent size> ]
             , [ <file type>             ]
             , [ <record length>         ]
             , [ <data block length>     ]
             , [ <key-sequenced params>  ]
             , [ <alternate key params>  ]
             , [ <partition params>      ]  )
```

where

<file name>, INT:ref, passed, [returned]

is an array providing the name of the disc file to be created in either of the following forms:

permanent disc files are created by specifying

```
<file name[0:3]>  is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
<file name[4:7]>  is <subvol name><blank fill>
<file name[8:11]> is <disc file name><blank fill>
```

temporary disc files are created by specifying

```
<file name[0:11]> is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
```

when CREATE completes, a <temporary file name> is returned in <file name[4:7]>. The temporary file can then be opened by passing <file name> to OPEN.

<primary extent size>, INT:value, passed

if present, is the size of the primary extent in 2048-byte units. The maximum <primary extent size> is 65535 (134,215,680 bytes). If omitted, a primary extent size of one (2048 bytes) is assigned.

<file code>, INT:value, passed

if present, is an application-defined file identification code (file codes 100 - 999 are reserved for use by Tandem Computers, Inc.). If omitted, a file code of zero is assigned.

<secondary extent size>, INT:value, passed

if present, is the size of the secondary extents in 2048-byte units (a file may have up to 15 secondary extents allocated). The maximum <secondary extent size> is 65535 (134,215,680 bytes). If omitted, the size of the primary extent is used for the secondary extent size.

<file type>, INT:value, passed

if present, specifies the type of the file to be created.

<file type>.<13:15> specifies the file structure:

0 = unstructured (default)
1 = relative
2 = entry-sequenced
3 = key-sequenced

<file type>.<12> 1 = specifies 'ODDUNSTR' for unstructured files. See Section 5, File Creation, for details.

<file type>.<12> 1 = specifies data compression for key-sequenced files. See Section 5, File Creation, for details.

<file type>.<11> 1 = specifies index compression for key-sequenced files. See Section 5, File Creation, for details.

<file type>.<10> 1 = File Label is written to disc each time the end-of-file is advanced. The effect of setting this parameter is the same as calling REFRESH after every operation that advances the end-of-file.

<file type>.<3:9> must be zero.

<file type>.<2> 1 = for systems with the Transaction Monitoring Facility (TMF), specifies this file is an audited file; for systems without TMF, must be zero.

<file type>.<0:1> must be zero

If <file type> is omitted, an unstructured file is created.

<record length>, INT:value, passed

if present, is the maximum length of the logical record in bytes. For structured files, the maximum record length is determined by the data block size. With a data block size of 4096, the maximum record length for entry-sequenced and relative files is 4072 bytes. With the same maximum data block size of 4096, the maximum record length for a key-sequenced file is 2035. For unstructured files, the maximum record length is 4096. If omitted, 80 is used for the <record length>.

<data block length>, INT:value, passed

for structured files, if present, is the length in bytes of each block of records in the file. <data block length> must be a multiple of 512 and can not be greater than 4096. <data block length> must be at least <record length> + 24. For a key-sequenced file <data block length> must be at least 2 * <record length> + 26. If omitted, 1024 is used for the <data block length>. Regardless of the specified record length and data block size, the maxixum number of records that can be stored in a data block is 511.

```
   <key-sequenced params>, INT:ref,                          passed

     is a three-word array containing parameters that describe
     this file.  This parameter is required for key-sequenced
     files and may be omitted for other file types.  The format
     for this array is shown in the "<key sequenced params>
     ARRAY" table which follows


   <alternate key params>, INT:ref,                          passed

     is an array containing parameters describing any alternate
     keys for this file.  This parameter is required if the file
     has alternate keys, otherwise it may be omitted or its first
     word must be zero.  The format for this array is shown in
     the "<alternate key params> ARRAY" table which follows.


   <partition params>, INT:ref,                              passed

     is an array containing parameters that describe this file if
     the file is a multi-volume file.  If the file is to span
     multiple volumes, then this parameter is required, otherwise
     this parameter may be omitted or its first word must be
     zero.  The format for this array is shown in the
     "<partition params> ARRAY" table which follows.


condition code settings:

   <  (CCL) indicates that the CREATE failed (call FILEINFO)
   =  (CCE) indicates that the file was created successfully
   >  (CCG) the device is not a disc


example

   CALL CREATE(filename,5,0);
   IF < THEN ...                    ! CREATE failed.
```

Considerations

- Disc Allocation at CREATE Time

  Execution of the CREATE procedure does not allocate any disc area; it only provides an entry in the volume's directory indicating that the file exists.

- Error Handling

  If the CREATE fails (i.e., condition code other than CCE returned), the reason for the failure can be determined by calling the FILEINFO procedure and passing -1 as the <file number> parameter.

- File Security

  The file is created with the default security associated with the process creator's access id. Security can be changed by opening the file and calling SETMODE or SETMODENOWAIT.

- Odd Unstructured Files

  When creating unstructured files, the value passed for <file type>.<12> determines how all subsequent read, write, and position operations to the file will work.

  If <file type>.<12> is passed as a 1, the values of <record specifier>, <read count>, and <write count> are all interpreted exactly. That is, a <write count> or <read count> of seven transfers exactly seven bytes.

  If <file type>.<12> is passed as a 0, the values of <record specifier>, <read count>, and <write count> are all rounded up to an even number before the operation is performed. That is, a <write count> or <read count> of seven is rounded up to eight and eight bytes are transferred.

- Key-Sequenced Parameter Array Format

The key-sequenced parameter array format is shown in Table 5-1.

Table 5-1. <key-sequenced params> ARRAY FORMAT

word:	
[0]	<key length>
[1]	<key offset>
[2]	<index block length>

where

<key length>, INT,

is the length, in bytes, of the record's primary key field

<key offset>, INT,

is the number of bytes from the beginning of the record where the primary key field starts.

<index block length>, INT,

is the length in bytes of each index block in the file. <index block length> must be a multiple of 512 and may not be greater than 4096. If zero is specified, then the value of <data block length> is used as the <index block length>

- Alternate Key Parameter Array Format

The alternate key parameter array format is shown in Table 5-2.

Table 5-2. <alternate key params> ARRAY FORMAT

	0 8
word: [0]	<nf alt files> <nk alt keys>
[1]	KEY DESCRIPTION FOR ALTERNATE KEY 0
	. . .
	KEY DESCRIPTION FOR ALTERNATE KEY nk - 1
[nk * 4 + 1]	FILE NAME OF KEY FILE 0
	. . .
	FILE NAME OF KEY FILE nf - 1

Key Description for key "k" consists of four words of the form:

	0 8
[k * 4 + 1]	<key specifier>
[k * 4 + 2]	<key attributes>
[k * 4 + 3]	<null value> <key length>
[k * 4 + 4]	<key file number>

→

Table 5-2. <alternate key params> ARRAY FORMAT (cont'd)

where

<nf alt files>, one-byte value,

specifies the number of alternate key files for this primary file.

<nk alt keys>, one-byte value,

specifies the number of alternate key fields in this primary file.

<key specifier>, INT,

is a two-byte value that uniquely identifies this alternate key field. This value is passed to the KEYPOSITION procedure when referencing this key field.

<key attributes>, INT,

describes the key.

where

<key attributes>.<0>: 1 = null value is specified. See "<null value>" below.

<key attributes>.<1>: 1 = key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insert is rejected with a "duplicate record" error.

<key attributes>.<2>: 1 = no automatic updating of this key is to be performed by ENSCRIBE.

<key attributes>.<3> must be zero.

<key attributes>.<4:15> = <key offset>. This specifies the the number of bytes from the beginning of the record where this key field starts.

→

Table 5-2. <alternate key params> ARRAY FORMAT (cont'd)

<null value>, one-byte value,

is used to specify a "null value" if <key attributes>.<0> = 1.

During an insertion (i.e., WRITE), if a null value is specified for an alternate key field and the null value is encountered in all bytes of this key field, ENSCRIBE does not enter the reference to the record into the alternate key file. (If the file is read via this alternate key field, records containing a null value in this field will not be found.)

During a deletion (i.e., WRITEUPDATE, <write count> = 0), if a null value is specified and the null value is encountered in all bytes of this key field within <buffer>, ENSCRIBE deletes the record from the primary file but does not delete the reference to the record in the alternate file.

<key length>, one-byte value,

that specifies the length, in bytes, of this key field.

<key file number>, INT,

is the relative number in the <alternate key params> array of this key's alternate key file. The first alternate key file's <key file number> = 0.

The File Name for file "f" consists of 12 words and begins at

[nk * 4 + 1 + f * 12]

and is of the form

```
<file name[0:3]>  is $<volume name><blank fill>
                  or \<system number><volume name><blank fill>
<file name[4:7]>  is <subvol name><blank fill>
<file name[8:11]> is <disc file name><blank fill>
```

- Partition Parameter Array Format

The partition parameter array format is shown in Table 5-3.

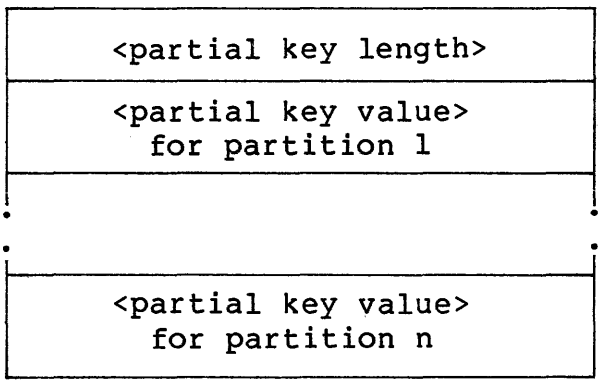
Table 5-3. <partition params> ARRAY FORMAT

	num words
<num extra partitions>	[1]
$<volume name> or \<sys num><volume name> for partition 1	[4]
$<volume name> or \<sys num><volume name> for partition 2	
. . .	
$<volume name> or \<sys num><volume name> for partition n	
<primary extent size> part 1	[1]
. . .	
<primary extent size> part n	
<secondary extent size> part 1	[1]
. . .	
<secondary extent size> part n	

→

Table 5-3. <partition params> ARRAY FORMAT (cont'd)

The following must be included in the <partition params> array for key-sequenced files and may be omitted for other file types:



[1]

where

<num extra partitions>, INT,

is the number of extra volumes (other than the one specified in the <file name> parameter) on which the file is to reside. The maximum value permitted is 15. Note that every other parameter in the partition array (except <partial key length>) must be specified <num extra partitions> times.

$<volume name> or
\<sys num><volume name>, eight bytes blank filled,

is the name of the disc volume (including "$" or "\") where the particular partition is to reside.

<primary extent size>, INT,

is the size of the primary extent for the particular partition.

<secondary extent size>, INT,

is the size of the secondary extents for the particular partition. Specifying zero results in the <primary extent size> value being used.

→

Table 5-3. <partition params> ARRAY FORMAT (cont'd)

The remaining parameters are required for key-sequenced files and may be omitted for all other file types.

<partial key length>, INT,

is the number of bytes of the primary key of a key-sequenced file that will be used to determine which partition of the file will contain a particular record. The minimum value for <partial key length> is one.

<partial key value>, INT,

for <partial key length> bytes, specifies the lowest key value that will be allowed for a particular partition.

Each <partial key value> in <partition params> must begin on a word boundary.

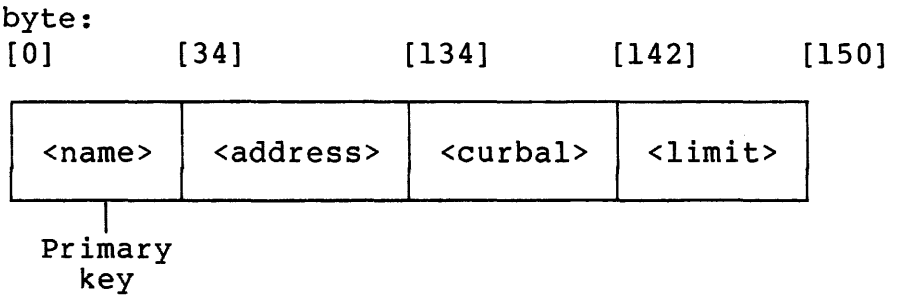
CREATION EXAMPLES

This section contains the following file creation examples.

1) Key-Sequenced file
2) Key-Sequenced file with alternate keys
3) Alternate Key file for programmatically created primary
4) Partitioned Relative file
5) Partitioned Key-Sequenced file

Example 1. Key-Sequenced File

To create a key-sequenced file for the following record,



using FUP, the following commands could be entered:

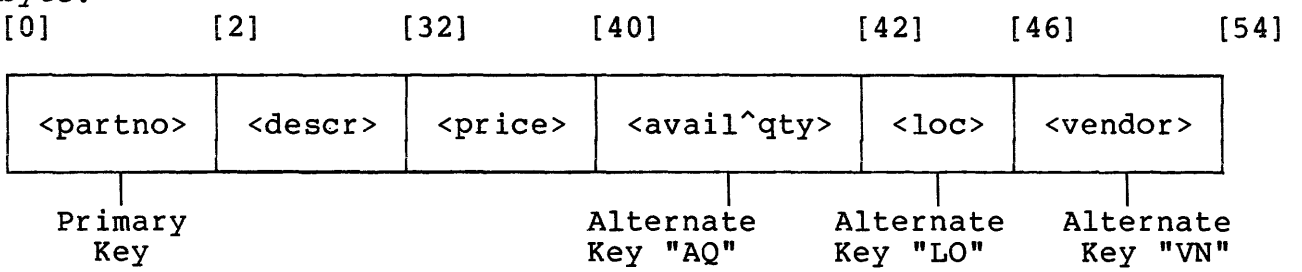
```
:FUP
GUARDIAN FILE UTILITY PROGRAM B06
-SET TYPE K
-SET CODE 1000
-SET EXT (16,1)
-SET REC 150
-SET BLOCK 2048
-SET COMPRESS
-SET IBLOCK 2048
-SET KEYLEN 34
-SHOW
    TYPE K
    CODE 1000
    EXT ( 16 PAGES, 1 PAGES )
    REC 150
    BLOCK  2048
    IBLOCK 2048
    KEYLEN 34
    KEYOFF 0
    DCOMPRESS, ICOMPRESS
-CREATE myfile
CREATED - $STORE1.SVOL1.MYFILE
```

Using the CREATE procedure, the following could be written in an application program:

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

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

LITERAL
     pri^extent = 16,           ! primary extent size = 16 * 2048.
     file^code = 1000,
     sec^extent = 1,            ! secondary extent size = 1 * 2048.
     file^type = %33,           ! file type = key-sequenced,
                                !             data and index
                                !             compression.
     rec^len = 150,             ! record length.
     data^block^len = 2048;     !

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

Example 2. Key-Sequenced File having Alternate Keys

To create a key-sequenced file for the following "INVENTORY" record,



Using FUP, the following commands could be entered:

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

Using the CREATE procedure, the following could be written in an application program:

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

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

                          ! key description for key 1.

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

                          ! key description for key 2.

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

                          ! key description for key 3.

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

                          ! key file name

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


LITERAL
     pri^extent = 32,           ! primary extent size = 32 * 2048.
     file^code = 1001,
     sec^extent = 8,            ! secondary extent size = 8 * 2048.
     file^type = %03,           ! file type = key-sequenced.
     rec^len = 54,              ! record length = 54.
     data^block^len = 4096;     ! data block length = 4096.
.
CALL CREATE (inv^filename, pri^extent, file^code,
             sec^extent, file^type, rec^len,
             data^block^len, pri^key, alt^keys);
IF < THEN ... ! error.
```

Note that the alternate key file must be created separately.

Example 3. Alternate Key File

The alternate key file for the preceding key-sequenced file is created automatically when FUP is used for file creation.

If the primary file is created programatically, the alternate file must be created in a separate operation. To create the alternate key file for the preceding key-sequenced file, the following is written in an application program:

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

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

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

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

Example 4. Relative, Partitioned File

To create a relative file with a record length of 128 bytes that spans four partitions,

Using FUP, the following commands could be entered:

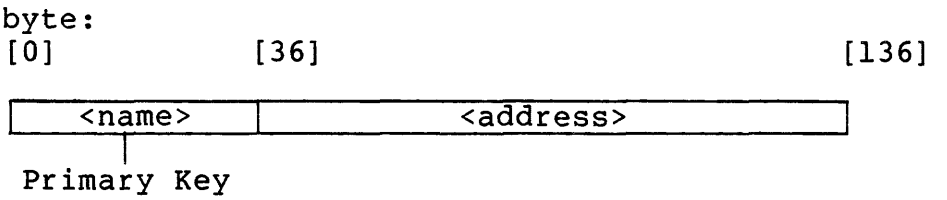
```
-SET TYPE R
-SET EXT (64,8)
-SET REC 128
-SET BLOCK 4096
-SET PART (1,$PART1,64,8)
-SET PART (2,$PART2,64,8)
-SET PART (3,$PART3,64,8)
-SHOW
    TYPE R
    EXT ( 64 PAGES, 8 PAGES )
    REC 128
    BLOCK  4096
    PART ( 1, $PART1, 64, 8 )
    PART ( 2, $PART2, 64, 8 )
    PART ( 3, $PART3, 64, 8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

Using the CREATE procedure, the following could be written in an application program:

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

LITERAL
     pri^extent = 64,           ! primary extent size = 64 * 2048.
     sec^extent = 8,            ! secondary extent size = 8 * 2048.
     file^type = %01,           ! file type = relative.
     rec^len = 128,             ! record length = 128.
     data^block^len = 4096;     ! data block length = 4096.

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

Example 5. Key-Sequenced, Partitioned File

The following is an example of partitioning for a key-sequenced file having the following record format:



The file resides on six volumes and is partitioned as follows:

<names> whose first two letters are in the range of		reside on
00	→	$PART0
DA	→	$PART1
HA	→	$PART2
LA	→	$PART3
PA	→	$PART4
TA	→	$PART5

Using FUP to create this file, the following commands would be entered to describe the partitioning:

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

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

```
                                    ! partition params array.
.partarray [0:36] := [ 5,           ! num extra partitions.
                       "$PART1  ",  ! vol name of first extra.
                       "$PART2  ",  ! vol name of second extra.
                       "$PART3  ",  ! vol name of third extra.
                       "$PART4  ",  ! vol name of fourth extra.
                       "$PART5  ",  ! vol name of fifth extra.
                       64,          ! pri ext for first extra.
                       64,          ! pri ext for second extra.
                       64,          ! pri ext for third extra.
                       64,          ! pri ext for fourth extra.
                       64,          ! pri ext for fifth extra.
                       8,           ! sec ext for first extra.
                       8,           ! sec ext for second extra.
                       8,           ! sec ext for third extra.
                       8,           ! sec ext for fourth extra.
                       8,           ! sec ext for fifth extra.
                       2,           ! partial key length = 2.
                       "DA",        ! key value for $PART1.
                       "HA",        ! key value for $PART2.
                       "LA",        ! key value for $PART3.
                       "PA",        ! key value for $PART4.
                       "TA" ];      ! key value for $PART5.
```

SECTION 6

ENSCRIBE FILE LOADING

The File Utility Program (FUP) can be used to load data into an existing file.

The FUP commands related to file loading are:

- LOAD

  Data is loaded into an existing structured disc file by means of the LOAD command. (The LOAD command does not load any related alternate key files.) For key-sequenced files, the percentage of data block and index block space to be left for future insertions (i.e., slack space) can be specified.

- LOADALTFILE

  Alternate key files are loaded with the alternate key records of a specified file by means of the LOADALTFILE command. A slack percentage can be specified. This command always performs a sort of the alternate key records before the actual load is performed.

- BUILDKEYRECORDS

  Because some systems may have insufficient disc space for the sort operation, the alternate file load operation can be separated into two steps by first performing a BUILDKEYRECORDS operations; the input to BUILDKEYRECORDS is the "primary" file for which the alternate key records are to be built; this output (which can be to a magnetic tape file) are the file's alternate key records. The key records on tape can then be loaded into the alternate key file by means of a COPY or LOAD command.

For an explanation of how to run the FUP program, refer to "Running FUP" in Section 5.

THE FUP LOAD COMMAND

The LOAD command is used to load data into a structured disc file without affecting any associated alternate key files. Existing data in the file being loaded is lost.

For key-sequenced files, the input records can be in sorted or unsorted order (unsorted is assumed unless the "SORTED" option is specified). Also for key-sequenced files, the percentage of slack space to be left for future insertions to the file can be specified.

The form of the LOAD command is:

```
LOAD <in file name> , <destination file name>

  [ , <load option> ] ...
```

where

<in file name>

specifies the file containing the records to be loaded.

<destination file name>

specifies an existing disc file to be loaded.

<load option>

for destination key-sequenced files, specifies whether or not the <in file> records are in sorted order (and, therefore, if a sort should be performed) and specifies load options; for any type destination file, specifies the <in file> format. <load option> is one of

```
{ SORTED                  }
{ <in option>             }
{ <key-seq option>        }
{ PARTOF $<volume name>   }
```

SORTED

for a key-sequenced destination file only, specifies that the records in the <in file> are in the destination file's key field order and therefore that a sort of the <in file> records should not be performed. If omitted, a sort of the <in file> records is performed before the loading of the <destination file> takes place.

→

<in option>

specifies the format and control of the in file.
<in option> is one of

RECIN <in record length>	
BLOCKIN <in block length>	
TRIM "<trim character>"	
EBCDICIN	
SHARE	
[NO] UNLOADIN	mag tape
[NO] REWINDIN	mag tape
SKIPIN <num eofs>	mag tape
REELS <num reels>	mag tape

See "FUP" in the "GUARDIAN Command Language and Utilities Manual" for an explanation of the above options.

<key-seq option>

specifies options pertaining to loading key-sequenced files.
<key-seq option> is one of

MAX <num records>
SCRATCH <scratch file name>
SLACK <percentage>
DSLACK <percentage>
ISLACK <percentage>

PARTOF $<volume name>

for key-sequenced, partitioned files only, specifies that only the partition designated by <destination file name> is to be loaded; $<volume name> is the volume where the primary partition of the <destination file> resides. (To load only the primary partition, the name of the primary partition is specified as the <destination file name>.)

example

```
LOAD $TAPE, ksfile, SORTED, DSLACK 10
```

Considerations

- Disc Space Requirement for the SORT Option

  If the <in file> records must be sorted, then during the sorting phase, disc space for both the sort scratch file and for the <destination file> must exist concurrently.

- Key-Sequenced File LOAD Options Explained

  The syntax for the <key-seq option> is:

```
{ MAX <num records>           }
{ SCRATCH <scratch file name> }
{ SLACK  <percentage>         }
{ DSLACK <percentage>         }
{ ISLACK <percentage>         }
```

  The following two options pertain to sorting the <in file> records. Therefore, if "SORTED" has been specified, these options can be ignored.

  - MAX <num records>

    specifies the number of records in the <in file>. <num records> is given as a value in the range of {0:2099999999}. The <num records> value is used to determine the size of the scratch file used by the SORT process. If omitted, 10,000 is used. This value need not be exact, but if given, should equal-to or greater-than then actual number of records in the <in file>.

  - SCRATCH <scratch file name>

    specifies a <file name> or $<volume name> to be used for temporary storage during the sorting phase. If omitted, a scratch file on the default volume is used.

  The following options specify the minimum percentage of space to be left in index and/or data blocks for future insertions. Note that if space is not available when an insertion is made, a "block split" will occur.

  - SLACK <percentage>

    specifies the minimum percentage of slack space in both index and data blocks. <percentage> is specified as a value in the range of {0:99}. If this option is omitted, 0 percentage will be left for slack space.

- DSLACK <percentage>

  specifies the minimum percentage of slack space in data blocks. If omitted, the "SLACK <percentage>" value is used.

- ISLACK <percentage>

  specifies the minimum percentage of slack space in index blocks. If omitted, the "SLACK <percentage>" value is used.

THE FUP LOADALTFILE COMMAND

The LOADALTFILE command is used to generate from a specified primary file the alternate key records associated with a designated alternate key file and load those records into that file. Slack space for future insertions can be specified.

The form of the LOADALTFILE command is:

```
LOADALTFILE <key file number> , <primary file name>

              [ , <key-seq option> ] ...
```

where

<key file number>

specifies the alternate key file to be loaded. <key file number> is an integer in the range of {0:255} indicating an alternate key file of the <primary file>. The alternate key file must already exist.

<primary file name>

specifies an existing primary file whose alternate key records are to generated and loaded into the file indicated by <key file number>.

<key-seq option>

specifies options pertaining to loading the alternate key file. <key-seq option> is one of

```
{ MAX <num records>           }
{ SCRATCH <scratch file name> }
{ SLACK  <percentage>         }
{ DSLACK <percentage>         }
{ ISLACK <percentage>         }
```

example

```
LOADALTFILE 0, ksfile, DSLACK 10
```

Considerations

- How LOADALTFILE Works

  Execution of this command causes a sort to be performed. The primary file is read sequentially via its primary key field. For each record read from the primary file, one or more alternate key records are generated and written to the SORT process. When the sort completes, the sorted records are read from the SORT process then loaded into the indicated alternate key file.

  Note that during the sorting phase, disc space for the sort scratch file and for the alternate key file must exist concurrently.

- LOADALTFILE and Null Alternate Key Fields

  Any "NULL" specification defined for a key field will be honored (i.e., an alternate key record will not be generated for a field having a null character defined if the field consists solely of the null character).

- LOADALTFILE and NO UPDATE Alternate Key Fields

  Any "NO UPDATE" specifications will be ignored.

- Key-Sequenced Options

  The syntax for the LOADALTFILE key-sequenced options is

```
{ MAX <num records>            }
{ SCRATCH <scratch file name> }
{ SLACK  <percentage>         }
{ DSLACK <percentage>         }
{ ISLACK <percentage>         }
```

  The following two options pertain to sorting the alternate key records generated from the <primary file>.

  - MAX <num records>

    specifies the number of records in the <source file>. <num records> is given as a value in the range of {0:2099999999}. The <num records> value is used to determine the size of the scratch file used by the SORT process. If omitted, 10,000 is used. This value need not be exact, but if given, should equal-to or greater-than then actual number of records in the <source file>.

- SCRATCH <scratch file name>

  specifies a <file name> or $<volume name> to be used for temporary storage during the sorting phase. If omitted, a scratch file on the default volume is used.

The following options specify the minimum percentage of space to be left in index and/or data blocks for future insertions. Note that if space is not available when an insertion is made, a "block split" will occur.

- SLACK <percentage>

  specifies the minimum percentage of slack space in both index and data blocks. <percentage> is specified as a value in the range of {0:99}. If this option is omitted, 0 percentage will be left for slack space.

- DSLACK <percentage>

  specifies the minimum percentage of slack space in data blocks. If omitted, the "SLACK <percentage>" value is used.

- ISLACK <percentage>

  specifies the minimum percentage of slack space in index blocks. If omitted, the "SLACK <percentage>" value is used.

THE FUP BUILDKEYRECORDS COMMAND

The BUILDKEYRECORDS command is used to generate the alternate key records for specified key fields of a specified structured disc file and write those records to a designated file (not necessarily the destination alternate key file). If the output file is not the destination file, the alternate key records generated by BUILDKEY-RECORDS are then loaded (possibly after being sorted) into the destination alternate key file by means of a LOAD command. (This is done in lieu of a direct load of the alternate file via a LOADALTFILE command when limited system resources do not permit such an operation.) Note that the output of BUILDKEYRECORDS can be the actual destination alternate key file; however, the alternate key loading will not be as efficient as using a LOAD command.

The form of the BUILDKEYRECORDS Command is:

```
BUILDKEYRECORDS <primary file name> , <out file name> ,
                    <key specifier list> [ , <out option> ] ...
```

where

<primary file name>

specifies an existing primary file whose alternate key records are to be generated. The primary file must have one or more alternate key fields defined.

<out file name>

specifies an existing file where the alternate key records generated by this command are to be written.

→

<key specifier list>

specifies one or more alternate key fields of <primary file name> whose corresponding alternate key records are to be generated. <key specifier list> is of the form

```
{ <key specifier>           }
{ ( <key specifier> , ... ) }
```

<key specifier>

is a two-byte value that uniquely identifies the alternate key field. It is specified as either

"[<c1>]<c2>"

a one- or two-character string within quotation marks (if <c1> is omitted, then <c1> is treated as a zero (0)); or

{-32768:65535}

<out option>

specifies the format and control of the out file. <out option> is one of

```
{ RECOUT <out record length>                    }
{ BLOCKOUT <out block length>                   }
{ PAD "<pad character>"                         }
{ EBCDICOUT                                     }
{ FOLD                                          }
{ [ NO ] UNLOADOUT                              } mag tape
{ [ NO ] REWINDOUT                              } mag tape
{ SKIPOUT <num eofs>                            } mag tape
```

See "FUP" in the "GUARDIAN Command Language and Utilities Manual" explanation of the above options.

example

```
BUILDKEYRECORDS myfile,$TAPE,("ab","cd")
```

Considerations

- How BUILDKEYRECORDS Works

  The execution of this command causes the primary file to be read sequentially via its primary key field. For each record read from the primary file, one or more alternate key records are generated and written to the <out file> (corresponding to the number of <key specifiers> specified). If more than one <key specifier> is specified, the corresponding alternate key file records are generated in order of the ASCII collating sequence of <key specifiers>.

- BUILDKEYRECORDS and NULL Alternate Key Fields

  Any "NULL" specification defined for a key field will be honored (i.e., an alternate key record will not be generated for a field having a null character defined if the field consists solely of the null character).

- BUILDKEYRECORDS and NO UPDATE Alternate Key Fields

  Any "NO UPDATE" specifications will be ignored.

- BUILDKEYRECORDS and UNIQUE Alternate Key Fields

  Any "UNIQUE" specifications are ignored; however, duplicate unique key values will be detected when the alternate key file is loaded.

FILE LOADING EXAMPLES

These examples illustrate file loading operations that require a sequence of FUP commands to perform. The examples in this section are:

1) Loading a Key-Sequenced file

2) Adding an Alternate Key to a file having Alternate Keys

3) Adding an Alternate Key to a file not having Alternate Keys

4) Reloading a single Partition of a Partitioned Key-Sequenced file

5) Loading a single Partition of a Partitioned Alternate Key file

Example 1. Load a Key-Sequenced File

The file is designated

```
$voll.svol.partfile.
```

It is a key-sequenced file having three partitions. The secondary partitions are

$vol2, the first secondary partition, and

$vol3, the second secondary partition.

Records having a primary key value in the range of zero up to, but not including "HA" are to be exist in the primary partition; records having a primary key value in the range of "HA" up to, but not including "RA" are to exist in the partition on the volume "$vol2"; records having a primary key value of "RA" or greater are to exist in the partition on the volume "$vol3".

The records to be loaded into this file are 128-bytes in length and are on tape in unsorted order (the tape is written with one record per block).

The FUP commands to perform this operation are

```
-VOLUME $voll.svol
-LOAD $TAPE, partfile
```

This reads the records from tape and sends them to the SORT process. When all records have been input, sorting begins. When the sort is finished, the records are read from the SORT process and loaded into the file according to the file's <partial key value> specifications. The data and index block slack percentage is zero (0).

Example 2. Add an Alternate Key to a File Having an Alternate Key

The file to which the key is to be added, the primary file, is designated

```
$voll.svol.prifile.
```

The primary file has one alternate key file designated

```
$voll.svol.altfile.
```

The alternate key records for the new key field will be added to this file.

The <key specifier> for the new key is "NM", the <key offset> in the record is four (4), the <key length> is twenty (20), a "null value" of " " (blank) is specified for the new key field.

The FUP commands to perform this operation are

```
-VOLUME $voll.svol
-ALTER prifile, ALTKEY ( "NM", KEYOFF 4, KEYLEN 20, NULL " " )
-LOADALTFILE 0, prifile, ISLACK 10
```

The LOADALTFILE command loads <key file number> zero (0) of "prifile", "$voll.svol.altfile" with the alternate key records for <key specifier> "NM" and for any other alternate keys defined for key file zero (0). An index block slack percentage of ten (10) is specified.

Example 3. Add an Alternate Key to a File Not Having Alternate Keys

The file to which the key is to be added, the primary file, is designated

```
$voll.svol.filea.
```

It is an entry-sequenced file.

The new alternate key file will be designated

```
$voll.svol.fileb.
```

The alternate key records for the new key field will be added to this file.

The <key specifier> for the new key is "XY", the <key offset> in the record is (0), the <key length> is ten (10).

The FUP commands to perform this operation are

```
-VOLUME $voll.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 "$voll.svol.fileb". The record length and key length are specifies as 16 bytes (2 for 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 references "fileb" as <alternate key file> 0, and contains the definition for the key field specified by <key specifier> "XY".

The LOADALTFILE command loads <key file number> zero (0) of "filea", "$voll.svol.fileb" with the alternate key records for <key specifier> "XY". An index block slack percentage of zero (0) is implied.

Example 4. Reload a Single Partition of Key-Sequenced, Partitioned File

The primary partition of the partitioned file is

$voll.svol.partfile

Partitions of "partfile" exist on

$vol2, the first secondary partition, and

$vol3, the second secondary partition.

The secondary partition (on "$vol2") is to be loaded.

The FUP commands to perform this operation are

```
-VOLUME $voll.svol
-SET LIKE $vol2.partfile
-SET NO PARTONLY
-CREATE temp
-DUP $vol2.partfile, temp, OLD, PARTONLY
-LOAD temp, $vol2.partfile, SORTED, PARTOF $voll
-PURGE temp
```

The SET and CREATE command create a file identical to "$vol2.svol.partfile" except that the file is designated a non-partitioned file by means of "NO PARTONLY".

The DUP command duplicates the data in the secondary partition "$vol2.svol.partfile" into "$voll.svol.temp".

The LOAD command reloads the secondary partition "$vol2.svol.partfile". Note that the "SORTED" option is specified because the records in the "temp" file are already in sorted order.

Example 5. Load a Single Partition of Partitioned, Alternate Key File

The primary file is designated

$voll.svol.prifile.

It is a key-sequenced file having a primary key field length of ten (10). It has three alternate key fields defined by the <key specifiers>

"F1", "F2", and "F3"

Each of these alternate key fields are ten (10) bytes in length.

All alternate key records are contained in one alternate key file that is partitioned over three volumes; each volume contains the alternate key records for one alternate key field (the <key specifier> for the alternate key field is also the <partial key value> for the secondary partitions).

The primary partition of the partitioned, alternate key file is

$voll.svol.altfile.

It contains the alternate key records for the <key specifier> "F1".

Partitions of the alternate key file "altfile" exist on

$vol2, the first secondary partition, and

$vol3, the second secondary partition.

"$vol2.svol.altfile" contains the alternate key records for the <key specifier> "F2". "$vol3.svol.altfile" contains the alternate key records for the <key specifier> "F3".

The alternate key records for the <key specifier> "F2" are to be loaded into "vol2.svol.altfile".

The FUP commands to perform this operation are

```
-VOLUME $voll.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 $voll.svol
-LOAD sortout, $vol2.altfile, SORTED, PARTOF $voll, RECIN 22,
   BLOCKIN 2200
-PURGE ! sortin, sortout
```

The CREATE commands create the disc file used as the output of BUILDKEYRECORDS (which is also the input to SORT) and the disc file to be used as the output of SORT.

The BUILDKEYRECORDS command generates the alternate key records for the <key specifier> "F2" of "prifile" and writes the records to "sortin". Record blocking is used to improve the efficiency of disc writes.

The Tandem-supplied SORT program is used to sort 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 the <key specifier> "F2". Note that the record blocking here is complementary to that used with BUILDKEYRECORDS.

APPENDIX A

SEQUENTIAL I/O PROCEDURES

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

Some characteristics of the Sequential I/O procedures are:

- All file types are accessed in a uniform manner.
  - File access characteristics, such as access mode, exclusion modes, and record size, are selected according to device type and the intended access.
  - The Sequential I/O procedures default characteristics are set to facilitate their most general use.
- Error recovery is automatic. All fatal errors cause the display of a comprehensive error message, all files to close and the process to abort. The automatic error handling and/or the display of error messages may be turned off. This allows the program to do the error handling.
- The characteristics of Sequential I/O operations can be altered at open time with the OPEN^FILE procedure. This is also possible before or after the open time with the SET^FILE procedure. Some optional characteristics are:
  - Record blocking/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.
- The Sequential I/O procedures can be used with the INITIALIZER procedure to make run-time changes. File transfer characteristics, such as record length, can be changed using the Command Interpreter ASSIGN command. (See "Interface with INITIALIZER and ASSIGN Messages".)
- The Sequential I/O procedures retain information about the files in file control blocks. There is one file control block (FCB) for each open file plus one common file control block which is linked to the other FCBs. (See FCB Structure.)

The Sequential I/O procedures and their functions are:

CHECK^BREAK	checks whether the break key was typed
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 Sequential I/O procedures
READ^FILE	reads from a file
SET^FILE	sets or alters file characteristics
TAKE^BREAK	enables the break key
WAIT^FILE	waits for the completion of an outstanding I/O operation
WRITE^FILE	writes to a file

The Sequential I/O procedures also contain a set of Defines and Literals that:

Allocate control block space (see "OPEN^FILE").

Specify open characteristics (see "OPEN^FILE").

Set file transfer characteristics (see "SET^FILE").

Check file transfer characteristics (see "CHECK^FILE").

Note that in the description of the procedure parameters, the commercial at symbol "@" is used to indicate the address of an object, not the object itself. For example, when specifying a file name to the SET^FILE procedure, the file name parameter should be passed as follows:

```
CALL SET^FILE ( in^file , ASSIGN^FILENAME , @buf );
```

where

@buf is the address of the array containing the name of the file to be opened.

SEQUENTIAL I/O PROCEDURES SOURCE FILES

The source file named $SYSTEM.SYSTEM.GPLDEFS is used with the Sequential I/O procedures. It provides the TAL Defines for allocating control block space, for assigning open characteristics to the file, and for altering and checking the file transfer characteristics. The TAL Literals for the Sequential I/O procedures error numbers are also included. This file must be referenced in the program's global area before any internal or external procedure declarations or within a procedure before any subprocedure declarations.

Like all other procedures in a TAL program, the Sequential I/O procedures must be declared before being called. These procedures are declared as external. The external declarations for these procedures are provided in a system file named $SYSTEM.SYSTEM.EXTDECS. A SOURCE compiler command specifying this file should be included in the source program following the global declarations but preceding the first call to one of these procedures:

```
<global declarations>

?SOURCE $SYSTEM.SYSTEM.GPLDEFS

?SOURCE $SYSTEM.SYSTEM.EXTDECS ( <names of procedures desired> )

<procedure declarations>
```

SIO: CHECK^BREAK Procedure

The CHECK^BREAK procedure tests whether the break key has been typed since the last CHECK^BREAK.

The call to CHECK^BREAK is

```
<state> := CHECK^BREAK ( { <common FCB> } )
                         { <file FCB>   }

where

   <state>, INT,

     indicates whether or not the break key has been typed.
     Values returned in <state> are:

       1 = break key typed and break is enabled
       0 = break key not typed or break is disabled

   <common FCB>, INT:ref,

   <file FCB>, INT:ref,

     identifies the file to be checked for break.  <common FCB>
     is allowed for convenience.
```

Example:

```
CALL TAKE^BREAK ( out^file );
WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
  BEGIN
    .
    .
    CALL WRITE^FILE ( out^file , buffer , count );
  END;

CALL GIVE^BREAK ( out^file );
```

CONSIDERATIONS

- If CR/LF on break is enabled, the default case, a carriage return/line feed sequence is executed on the terminal where break is typed.

The CHECK^FILE procedure checks the file characteristics.

The call to CHECK^FILE is

```
<retval> := CHECK^FILE ( { <common FCB> } , <operation> )
                         { <file FCB>   }

where

   <retval>, INT,

     is the value returned for the requested operation.

   <common FCB or file FCB>, INT:ref,

     identifies which file is checked. A <common FCB> can be
     used for certain types of checks;  a <common FCB> must be
                                                        must
     used for the checks FILE^BREAKHIT, FILE^ERRORFILE,
     and FILE^TRACEBACK.  Specifying an improper FCB causes
     an error indication.

   <operation>, INT:value,

     specifies which file characteristic is checked.  The
     <operation>s and their associated <retval>s are:


     <operation> = FILE^ABORT^XFERERR,    (file must be open)
     <retval> := <bit value>

       returns: 0  if the process is not to abort upon
                   detection of a fatal error in the file or
                1  if the process is to abort.

     <operation> = FILE^ASSIGNMASK1,
     <retval> := <high-order word of ASSIGN fieldmask>

       returns the high-order word of the ASSIGN message
       "fieldmask" in the FCB.  This value generally has
       meaning only after being set by the INITIALIZER
       procedure.
```

→

```
<operation> = FILE^ASSIGNMASK2,
<retval> := <low-order word of ASSIGN fieldmask>

  returns the low-order word of the ASSIGN message
  "fieldmask" in the FCB.  This value generally has
  meaning only after being set by the INITIALIZER
  procedure.

<operation> = FILE^BLOCKBUFLEN,
<retval> := <block buffer length>

  returns a count of the number of bytes used for blocking.

<operation> = FILE^BREAKHIT,
<retval> := <state of the break hit bit>

  returns: 0  if the break hit bit is equal to zero in
              the FCB or
           1  if the break hit bit is equal to one in the
              FCB.

  The break hit bit is an internal indicator normally
  used only by the Sequential I/O procedures.

  When using the break handling procedures, do not use
  FILE^BREAKHIT to determine if the break key has been
  typed.  Instead, the CHECK^BREAK procedure must be
  called.

<operation> = FILE^BWDLINKFCB,
<retval> := <backward-link-pointer>

  returns the address of the FCB pointed to by the backward
  link pointer within the FCB.  This indicates the
  linked-to FCB's that need to be checkpointed after an
  OPEN^FILE or CLOSE^FILE.

<operation> = FILE^CHECKSUM,          (file must be open)
<retval> := <checksum word>

  returns the value of the checksum word in the FCB.


<operation> = FILE^CREATED,           (file must be open)
<retval> := <state of the created bit>

  returns: 0  if a file was not created by OPEN^FILE or
           1  if a file was created by OPEN^FILE.
```

→

```
<operation> = FILE^COUNTXFERRED,      (file must be open)
<retval> := <count transferred>
```

returns a count of the number of bytes transferred in the latest physical I/O operation.

```
<operation> = FILE^CRLF^BREAK,        (file must be open)
<retval> := <state of cr/lf break bit>
```

returns: 0 if no carriage return and line feed sequence is to be issued to the terminal upon break detection or
1 if this sequence is to be issued.

```
<operation> = FILE^DUPFILE,           (file must be open)
<retval> := @<dupfile FCB>
```

returns the word address of the duplicate file FCB. A zero is returned if there is no duplicate file.

```
<operation> = FILE^ERROR,             (file must be open)
<retval> := <error>
```

returns the error number of the latest error that occurred within the file.

```
<operation> = FILE^ERRORFILE,
<retval> := @<error file FCB>
```

returns the word address within the FCB of the reporting error file. A zero is returned if there is none.

```
<operation> = FILE^ERROR^ADDR,
<retval> := @<error>
```

returns the word address within the FCB of where the error code is stored.

→

```
<operation> = FILE^FILEINFO          (file must be open)
<retval> := <file info>

  <file info>.<0:3> = file type, 0 = unstructured,
                                 1 = relative,
                                 2 = entry-sequenced,
                                 3 = key-sequenced,
                                 4 = edit,
                                 8 = odd-unstructured.

            .<4:9>   = dev type.
            .<10:15> = dev subtype.
```

File types 1-3 are described in Enscribe Disc Files, ENSCRIBE Data Base Record Manager Programming Manual, in the File Management section.

The device type and device subtype are described in the GUARDIAN Programming Manual, DEVICEINFO Procedure.

```
<operation> = FILE^FILENAME^ADDR,
<retval> := @<filename>
```

returns the word address within the FCB of the physical filename.

```
<operation> = FILE^FNUM,             (file must be open)
<retval> := <file number>
```

returns the file number.

```
<operation> = FILE^FNUM^ADDR,
<retval> := @<file number>
```

returns the word address within the FCB of the file number. If the file is not open, the file number is -1.

```
<operation> = FILE^FWDLINKFCB,
<retval> := <forward-link-pointer>
```

returns the address of the FCB pointed to by the forward link pointer within the FCB. This value indicates the linked-to FCB's that need to be checkpointed after an OPEN^FILE or CLOSE^FILE.

```
<operation> = FILE^LOGICALFILENAME^ADDR
<retval> := @<logical filename>

  returns the word address within the FCB of the logical
  filename.  The logical filename is encoded as follows:

    byte numbers

    [0]  [1]                  [8]
    <len><logical file name>

    <len> is the length of the logical file name in bytes
    {0:7}.

<operation> = FILE^LOGIOOUT,          (file must be open)
<retval> := <state of the logioout bit>

  returns:  0 to indicate there is no logical I/O
              outstanding.
            1 if a logical read is outstanding.
            2 if a logical write is outstanding.

<operation> = FILE^PHYSIOOUT,         (file must be open)
<retval> := <state of the physioout bit>

  returns:  0 to indicate there is no outstanding physical
              I/O operation.
            1 if a physical I/O is outstanding.

<operation> = FILE^PRIEXT,
<retval> := <primary extent size>

  returns the file's primary extent size in pages.

<operation> = FILE^PRINT^ERR^MSG,     (file must be open)
<retval> := <state of print errmsg bit>

  returns:  0 if no error message is to be printed upon
              detection of a fatal error in the file.
            1 if an error message is to be printed.

<operation> = FILE^PROMPT,            (file must be open)
<retval> := <interactive prompt character>

  returns the interactive prompt character for the file
  in <9:15>.
```

→

```
<operation> = FILE^RCVEOF,              (file must be open)
<retval> := <state of rcveof bit>

  returns:  0 if the user does not get an end-of-file
              (EOF) indication, when the last process[-pair]
              having this process open, closes it.
            1 if the user does get an EOF indication, when
              this process closes.

<operation> = FILE^RCVOPENCNT,          (file must be open)
<retval> := <$RECEIVE opener count>

  returns a count of current openers of this process {0:2}.
  At any given moment openers are limited to a single
  process[-pair].

<operation> = FILE^RCVUSEROPENREPLY, (file must be open)
<retval> := <state of the rcv-user-open-reply bit>

  returns:  0 if the Sequential I/O procedures are to
              reply to the open messages ($RECEIVE file).
            1 if the user is to reply to the open messages.

<operation> = FILE^READ^TRIM,           (file must be open)
<retval> := <state of the read trim bit>

  returns:  0 to indicate the trailing blanks are not
              trimmed off the data read from this file.
            1 if the trailing blanks are trimmed.

<operation> = FILE^RECORDLEN,
<retval> := <record length>

  returns the logical record length.

<operation> = FILE^SECEXT,
<retval> := <secondary extent size>

  returns the file's secondary extent size in pages.
```

→

```
<operation> = FILE^LOGICALFILENAME^ADDR
<retval> := @<logical filename>

  returns the word address within the FCB of the logical
  filename.  The logical filename is encoded as follows:

    byte numbers

    [0]  [1]                  [8]
    <len><logical file name>

    <len> is the length of the logical file name in bytes
    {0:7}.

<operation> = FILE^LOGIOOUT,          (file must be open)
<retval> := <state of the logioout bit>

  returns:  0 to indicate there is no logical I/O
              outstanding.
            1 if a logical read is outstanding.
            2 if a logical write is outstanding.

<operation> = FILE^PHYSIOOUT,         (file must be open)
<retval> := <state of the physioout bit>

  returns:  0 to indicate there is no outstanding physical
              I/O operation.
            1 if a physical I/O is outstanding.

<operation> = FILE^PRIEXT,
<retval> := <primary extent size>

  returns the file's primary extent size in pages.

<operation> = FILE^PRINT^ERR^MSG,     (file must be open)
<retval> := <state of print errmsg bit>

  returns:  0 if no error message is to be printed upon
              detection of a fatal error in the file.
            1 if an error message is to be printed.

<operation> = FILE^PROMPT,            (file must be open)
<retval> := <interactive prompt character>

  returns the interactive prompt character for the file
  in <9:15>.
```

→

```
<operation> = FILE^RCVEOF,              (file must be open)
<retval> := <state of rcveof bit>

  returns:  0 if the user does not get an end-of-file
              (EOF) indication, when the last process[-pair]
              having this process open, closes it.
            1 if the user does get an EOF indication, when
              this process closes.

<operation> = FILE^RCVOPENCNT,          (file must be open)
<retval> := <$RECEIVE opener count>

  returns a count of current openers of this process {0:2}.
  At any given moment openers are limited to a single
  process[-pair].

<operation> = FILE^RCVUSEROPENREPLY, (file must be open)
<retval> := <state of the rcv-user-open-reply bit>

  returns:  0 if the Sequential I/O procedures are to
              reply to the open messages ($RECEIVE file).
            1 if the user is to reply to the open messages.

<operation> = FILE^READ^TRIM,           (file must be open)
<retval> := <state of the read trim bit>

  returns:  0 to indicate the trailing blanks are not
              trimmed off the data read from this file.
            1 if the trailing blanks are trimmed.

<operation> = FILE^RECORDLEN,
<retval> := <record length>

  returns the logical record length.

<operation> = FILE^SECEXT,
<retval> := <secondary extent size>

  returns the file's secondary extent size in pages.
```

```
<operation> = FILE^SEQNUM^ADDR
<retval> := @<sequence number>
```

returns the word address within the FCB of an INT (32) sequence number. This is the line number of the last record of an edit file. For a non-edit file this is the sequence number of the last record multiplied by 1000.

```
<operation> = FILE^SYSTEMMESSAGES,    (file must be open)
<retval> := <system message mask>
```

returns a mask word indicating which system messages the user handles directly. See SET^FILE for the format. A zero indicates that the Sequential I/O procedures handle all system messages.

```
<operation> = FILE^TRACEBACK,
<retval> := <state of traceback bit>
```

returns: 0 if the P-relative address should not be appended to all SIO error messages
1 if the P-relative address should be appended to all SIO error messages

```
<operation> = FILE^USERFLAG,
<retval> := <user flag>
```

returns the user flag word. (See SET^FILE procedure, SET^USERFLAG operation.)

```
<operation> = FILE^USERFLAG^ADDR
<retval> := @<user flag>
```

returns the word address within the FCB of the user flag word.

```
<operation> = FILE^WRITE^FOLD,        (file must be open)
<retval> := <state of the write-fold bit>
```

returns: 0 if records longer than the logical record length are truncated.
1 if long records are folded.

→

```
      <operation> = FILE^WRITE^PAD,        (file must be open)
      <retval> := <state of write-pad bit>

        returns:  0 if a record shorter than the logical record
                    length is not padded with trailing blanks
                    before it is written to the file.
                  1 if a short record is padded with trailing
                    blanks.

      <operation> = FILE^WRITE^TRIM,       (file must be open)
      <retval> := <state of the write-trim bit>

        returns:  0 if trailing blanks are not trimmed from
                    data written to the file.
                  1 if trailing blanks are trimmed.

  examples:

     INT .infile^name;

     @infile^name := CHECK^FILE ( infile , FILE^FILENAME^ADDR);


     INT .infnum;

     @infnum := CHECK^FILE ( infile , FILE^FNUM^ADDR);


     IF ( error := CHECK^FILE ( infile , FILE^ERROR ) ) THEN..
```

CONSIDERATIONS

- During the execution of this procedure the detection of any error causes the display of an error message and the process is aborted.

The CLOSE^FILE procedure is used to close a file.

The call to CLOSE^FILE is

```
{ CALL       } CLOSE^FILE ( { <common FCB> }
{ <error> := }              { <file FCB>   }
                              , <tape disposition> )
```

where

<error>, INT,

is either a file management or Sequential I/O procedure error number indicating the outcome of the close. In any case, the file is closed.

If the abort-on-error mode, the default, is in effect, the only possible value for <error> is zero.

<common FCB>, INT:ref,

identifies all files to be closed. If the break for any file is currently enabled, it is disabled.

<file FCB>, INT:ref,

identifies the file to be closed. If the break for the file is currently enabled, it is disabled.

<tape disposition>, INT:value

specifies mag tape disposition

where

<tape disposition>.<13:15>

0 = rewind, unload, do not wait for completion
1 = rewind, take offline, do not wait for completion
2 = rewind, leave online, do not wait for completion
3 = rewind, leave online, wait for completion
4 = do not rewind, leave online

example

```
CALL CLOSE^FILE ( common^fcb );
CALL CLOSE^FILE ( rcv^file );
```

CONSIDERATIONS

- Edit files or files that are open with write access and blocking capability must be closed with the CLOSE^FILE procedure or the data is lost.

The GIVE^BREAK procedure returns break to the previous owner, the process that had the break enabled before the last call to TAKE^BREAK.

The call to GIVE^BREAK is

```
CALL GIVE^BREAK  (  { <common FCB> }  )
                    { <file FCB>   }

where

   <common FCB>, INT:ref,

   <file FCB>, INT:ref,

     identifies the file receiving break.  <common FCB> is
     allowed for convenience.  If break is not enabled, this
     call is ignored.


example:

   CALL TAKE^BREAK ( out^file );
   WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
     BEGIN
       .
       .
       CALL WRITE^FILE ( out^file , buffer , count );
     END;

  CALL GIVE^BREAK ( out^file );
```

The OPEN^FILE procedure permits access to a file with the other Sequential I/O procedures.

The call to OPEN^FILE is

```
{ CALL       } OPEN^FILE ( <common FCB>
{ <error> := }
                         ,   <file FCB>
                         , [ <block buffer>        ]
                         , [ <block buffer length> ]
                         , [ <flags>               ]
                         , [ <flags mask>          ]
                         , [ <max record length>   ]
                         , [ <prompt char>         ]
                         , [ <error file FCB>      ] )
```

where

<error>, INT,

is a file management or Sequential I/O procedure error indicating the outcome of the operation.

If the abort-on-open-error mode is in effect, the only possible value of <error> is zero.

<common FCB>, INT:ref,

is an array of FCBSIZE words for use by the Sequential I/O procedures. Only one <common FCB> is used per process. This means the same data block is passed to all OPEN^FILE calls. The first word of <common FCB> must be initialized to zero before the first OPEN^FILE call following a process startup.

<file FCB>, INT:ref,

is an array of FCBSIZE words for use by the Sequential I/O procedures. The <file FCB> uniquely identifies this file to the other Sequential I/O procedures. The <file FCB> must be initialized with the name of the file to be opened before the OPEN^FILE call is made.

See "Initializing the File FCB" following the description of the FCB Structure.

→

<block buffer>, INT:ref,

(optional) is an array used for record blocking and deblocking. No blocking is performed if <block buffer> or <block buffer length> is omitted, or if the <block buffer length> is insufficient according to the record length for the file, or if readwrite access is indicated.

Blocking is performed when this parameter is supplied, the <block buffer> is of sufficient length, as indicated by the <block buffer length> parameter, and blocking is appropriate for the device.

The block buffer must be located within 'G' [0:32767] of the data area.

<block buffer length>, INT:value,

(optional) indicates the length, in bytes, of the <block buffer>. This length must be able to contain at least one logical record. For an edit file, the minimum length on read is 144 bytes; on write, the minimum length is 1024 bytes.

<flags>, INT(32):value,

(optional) is used in conjunction with the <flags mask> parameter to set file transfer characteristics. If omitted, all positions are treated as zero. The bit fields in <flags> are defined in Appendix A. These literals may be combined using signed addition, since bit 0 is not used.

ABORT^OPENERR,

abort-on-open error, defaults to on. If on, and a fatal error occurs during the OPEN^FILE, all files are closed and the process abends. If off, the file system or Sequential I/O procedure error number is returned to the caller.

ABORT^XFERERR,

abort-on-data-transfer error, defaults to on. If on, and a fatal error occurs during a data transfer operation, like a call to any Sequential I/O procedure except OPEN^FILE, all files are closed and the process abends. If off, the file system or the Sequential I/O procedure error number is returned to the caller.

→

PRINT^ERR^MSG,

print-error-message, defaults to on. If on, and a fatal error occurs, an error message will be displayed on the error file. This is the home terminal unless otherwise specified.

AUTO^CREATE,

auto create, defaults to on. If on, and open access is write, a file will be created provided one is not already there. If no file code has been assigned, or if the file code is 101, and a block buffer of sufficient size is provided, an edit file is created. The default extent sizes are 4 pages for the primary extent and 16 pages for the secondary extent.

MUSTBENEW,

file must-be-new, defaults to off. This applies only if AUTO^CREATE is specified.

PURGE^DATA,

purgedata, defaults to off. If on, and open access is write, the data will be purged from the file after the open. If off, the data is appended to the existing data.

AUTO^TOF,

auto top-of-form, defaults to on. If on, the file is open with write access and is a line printer or process, a page eject is issued to the file within the OPEN^FILE procedure.

NOWAIT,

nowait I/O, defaults to off or wait I/O. If on, nowait I/O is in effect.

BLOCKED,

non-disc blocking, defaults to off. A block buffer of sufficient length must also be specified.

VAR^FORMAT,

variable length records, defaults to off, or fixed length records. If on, the maximum record length for variable length records is 254 bytes.

READ^TRIM,

read trailing blank trim, defaults to on. If on, the <count read> parameter does not account for trailing blanks.

WRITE^TRIM,

write trailing blank trim, defaults to on. If on, trailing blanks are trimmed from the output record before being written to the file.

WRITE^FOLD,

write fold, defaults to on. If on, writes that exceed the record length cause multiple logical records to be written. If off, writes that exceed the record length are truncated to record length bytes; no error message or warning is given.

WRITE^PAD,

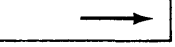
write blank pad, defaults to on for disc fixed length records and off for all other files. If on, writes of less than record length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.

CRLF^BREAK,

carriage return-line feed (CR/LF) on break, defaults to on. If on and break is enabled, a CR/LF feed will be written to the terminal when break is typed.

<flags mask>, INT(32):value,

(optional) specifies which bits of the flag field are used to alter the file transfer characteristics. The characteristic to be altered is indicated by entering a one in the bit position corresponding to the <flags> parameter. A zero indicates the default setting is used. When omitted, all positions are treated as zeros.



<max record length>, INT:value,

(optional) specifies the maximum record length for records within this file. If omitted, the <max record length> is 132. The open is aborted with an SIOERR^INVALIDRECLENGTH, error 520, if the file's record length exceeds the <max record length> and <max record length> is not zero. If the <max record length> is zero, then any record length is permitted.

<prompt char>, INT:value,

(optional) is used to set the interactive prompt character for reading from terminals or processes. When not supplied, the prompt defaults to "?". The prompt character is limited to seven bits, <9:15>.

<error file FCB>, INT:ref,

(optional) specifies a file where error messages are displayed for all files. Only one error reporting file is allowed per process. The file specified in the latest open is the one used. Omitting this parameter does not alter the current error reporting file setting.

The error reporting file is used for reporting errors when possible. If this file cannot be used or the error is with the error reporting file, the default error reporting file will be used. This is the home terminal.

If the error reporting file is not open when needed, it is opened only for the duration of the message printing then closed. Note that the <error file FCB> must be initialized. See "Initializing the File FCB".

Considerations

- If AUTO^TOF is on, a top-of-form control operation is performed to the file when the file being opened is a process or a line printer and write or readwrite access is specified.
- If the file is an edit file or when blocking is specified, either read or write access must be specified for the open to succeed. Readwrite access is not permitted.

- When using OPEN^FILE to access a temporary disc file, AUTO^CREATE must be disabled; otherwise the OPEN^FILE call results in a file management error 13.
- All files opened with the OPEN^FILE procedure are opened with a sync depth of one. One is the only possible sync depth; no other can be set.

Example:

```
LITERAL  prompt= ">",              !prompt character
         buffer^size = 144;        !minimum edit file buffer size

INT      error,
        .common^fcb [ 0:FCBSIZE-1 ] := 0,
        .in^file [ 0:FCBSIZE-1 ] := 0,
        .in^filename [0:11 ] := [ "$VOLUME SUBVOL  FILENAME" ],
        .buffer [ 0:buffer^size >> 1 ];

INT(32)  flags := 0D,
         flags^mask := ABORT^OPENERR; !return control on error

CALL SET^FILE ( in^file , INIT^FILEFCB );
CALL SET^FILE ( in^file , ASSIGN^FILENAME, @in^filename );
IF ( error := OPEN^FILE  ( common^fcb ,
                           in^file ,
                           buffer ,
                           buffer^size ,
                           flags ,
                           flags^mask ,
                           prompt ) ) THEN
   BEGIN
   !
   ! handle open error here
   !
   END;
```

The READ^FILE procedure is used to read a file sequentially. The file must be open with read or readwrite access.

The call to READ^FILE is

```
{ CALL       } READ^FILE (     <file FCB>
{ <error> := }
                             ,   <buffer>
                             ,   <count read>
                             , [ <prompt count>   ]
                             , [ <max read count> ]
                             , [ <no wait>        ] )
```

where

<error>, INT,

is a file system or Sequential I/O procedure error indicating the outcome of the read.

If abort-on-error mode is in effect, the only possible values for <error> are:

0 = no error

1 = end-of-file

6 = system message (only if user requested system messages, by SET^SYSTEMMESSAGES)

111 = operation aborted because of break (if break is enabled)

If <no wait> is not zero, and if abort-on-error is in effect, the only possible value for <error> is zero.

<file FCB>, INT:ref,

identifies the file to be read.

<buffer>, INT:ref,

is where the data is returned. The <buffer> must be located within 'G' [0:32767] process data area.

→

<count read>, INT:ref,

(optional if <no wait> is not zero) is the count of the number of bytes returned to <buffer>. If <no wait> is not zero, then this parameter has no meaning and can be omitted. The count is then obtained in the call to WAIT^FILE.

<prompt count>, INT:value,

(optional) is a count of the number of bytes in <buffer>, starting with element zero, to be used as an interactive prompt for terminals or interprocess files. If omitted, the interactive prompt character defined in OPEN^FILE is used.

<max read count>, INT:value,

(optional) specifies the maximum number of bytes to be returned to <buffer>. If omitted or if <max read count> exceeds the file's logical record length, the logical record length is used for this value.

<no wait>, INT:value,

(optional) indicates whether or not to wait for the I/O to complete in this call. If omitted or <no wait> is zero, then wait is indicated. If <no wait> is not zero, the I/O must be completed in a call to WAIT^FILE.

example

```
WHILE NOT ( error := READ^FILE ( in^file , buffer ,
                                 count ) ) DO
  BEGIN
    .
    .
  END;
```

CONSIDERATIONS

- If the file is a terminal or process a WRITEREAD operation will be performed using the interactive prompt character or <prompt count> character from <buffer>.

The SET^FILE procedure alters file characteristics and checks the old value of those characteristics being altered.

The call to SET^FILE is

```
{ CALL       } SET^FILE ( { <common FCB> } , <operation>
{ <error> := }            { <file FCB>   }
                                          , [ <new value> ]
                                          , [ <old value> ] )
```

where

<error>, INT,

is a file system or Sequential I/O procedure error indicating the outcome of the SET^FILE.

If abort-on-error mode is in effect, the only possible value for <error> is zero.

<common FCB>, INT:ref,

identifies those files whose characteristics are to be altered. The <common FCB> can be used for certain operations; it must be used for the operations SET^BREAKHIT, SET^ERRORFILE, and SET^TRACEBACK. If an improper FCB is specified, an error is indicated.

<file FCB>, INT:ref,

identifies the file whose characteristics are to be altered. If an improper FCB is specified, an error is indicated.

<operation>, INT:value,

specifies the file characteristic to be altered. See "List of SET^FILE Operations".

<new value>, INT:value,

specifies a new value for the specified <operation>. This may be optional depending on the <operation> desired.

→

```
    <old value>, INT:ref,

      is a variable in which the current value for the specified
      <operation> is returned.  This can vary from 1 word to 12
      words and is useful in saving this value for reset later.
      If <old value> is omitted, the current value is not
      returned.


LIST OF SET^FILE OPERATIONS

This is a list of the file characteristics which can be altered
by the SET^FILE procedure.


<operation> = ASSIGN^BLOCKBUFLEN (or, ASSIGN^BLOCKLENGTH)
<new value> = <new block length>   (optional, file must be closed)
<old value> := <block length>                                (optional)

  specifies the block length (in bytes) for the file.

<operation> = ASSIGN^FILECODE
<new value> = <new file code>      (optional, file must be closed)
<old value> := <file code>                                   (optional)

  specifies the file code for the file.

<operation> = ASSIGN^FILENAME
<new value> = @<file name>         (optional, file must be closed)
<old value> := <file name> FOR 12 words                      (optional)

  specifies the physical name of the file to be opened.  This
  operation is not used when the INITIALIZER procedure is called
  to initialize the file control blocks.

example

    CALL SET^FILE ( in^file , ASSIGN^FILENAME , @in^filename );
```

→

```
<operation> = ASSIGN^LOGICALFILENAME
<new value> = @<logical file name> (optional, file must be closed)
<old value> := <logical file name> FOR 4 words          (optional)

  specifies the logical name of the file to be opened.  The
  <logical file name> must be encoded as follows:

  byte numbers

  [0]  [1]                [8]
  <len><logical filename>

  <len> is the length of logical file name {0:7}.

<operation> = ASSIGN^OPENACCESS
<new value> = <new open access>    (optional, file must be closed)
<old value> := <open access>                            (optional)

  specifies the open access for the file.  The following literals
  are provided for <open access>.

    READWRITE^ACCESS    (0)
    READ^ACCESS         (1)
    WRITE^ACCESS        (2)

  Even if READ^ACCESS is specified, SIO actually opens the file
  with READWRITE^ACCESS to facilitate interactive I/O.

<operation> = ASSIGN^OPENEXCLUSION
<new value> = <new open exclusion> (optional, file must be closed)
<old value> := <open exclusion>                         (optional)

  specifies the open exclusion for the file.  The following
  literals are provided for <open exclusion>:

    SHARE           (0)
    EXCLUSIVE       (1)
    PROTECTED       (3)

<operation> = ASSIGN^PRIEXT (or, ASSIGN^PRIMARYEXTENTSIZE)
<new value> = <new pri ext size>   (optional, file must be closed)
<old value> := <pri ext size>                           (optional)

  specifies the primary extent size (in units of 2048-byte blocks)
  for the file.
```

→

```
<operation> = ASSIGN^RECORDLEN (or, ASSIGN^RECORDLENGTH)
<new value> = <new record length>  (optional, file must be closed)
<old value> := <record length>                              (optional)
```

specifies the logical record length (in bytes) for the file. For defaults, see step 6 under "Initializing the File FCB."

```
<operation> = ASSIGN^SECEXT (or, ASSIGN^SECONDARYEXTENTSIZE)
<new value> = <new sec ext size>   (optional, file must be closed)
<old value> := <sec ext size>                               (optional)
```

specifies the secondary extent size (in units of 2048-byte blocks) for the file.

```
<operation> = INIT^FILEFCB                        (file must be closed)
<new value> = must be omitted
<old value> = must be omitted
```

specifies that the <file FCB> be initialized. This operation is not used when the INITIALIZER procedure is called to initialize the file control blocks.

example

```
    CALL SET^FILE ( common^fcb , INIT^FILEFCB );
    CALL SET^FILE ( in^file , INIT^FILEFCB );
```

```
<operation> = SET^ABORT^XFERERR                    (file must be open)
<new value> = <new state>                                   (optional)
<old value> := <state>                                      (optional)
```

Sets/clears abort on transfer error for the file. If on, and a fatal error occurs during a data transfer operation, such as a call to any Sequential I/O procedure except OPEN^FILE, all files are closed and the process abends. If off, the file management or Sequential I/O procedure error number is returned to the caller.

```
<operation> = SET^CHECKSUM
<new value> = <new checksum word>
<old value> := <checksum word in FCB>
```

Sets/clears checksum word in the FCB. This is useful after modifying an FCB directly (i.e., without using the SIO procedures).

```
<operation> = SET^CRLF^BREAK                    (file must be open)
<new value> = <new state>                                (optional)
<old value> := <state>                                   (optional)
```

Sets/clears carriage return/line feed on break for the file. If on, a CR/LF is executed on the terminal when the break key is typed.

```
<operation> = SET^DUPFILE                       (file must be open)
<new value> = @<new dup file FCB>                        (optional)
<old value> := @<dup file FCB>                           (optional)
```

specifies a duplicative file for the file. This is a file where data read from <file FCB> is printed. Defaults to no duplicative file.

example

```
    CALL SET^FILE (in^file, SET^DUPFILE, @out^file);
```

```
<operation> = SET^ERRORFILE
<new value> = @<new error file FCB>                      (optional)
<old value> := @<error file FCB>                         (optional)
```

Sets error reporting file for all files. Defaults to home terminal. If the error reporting file is not open when needed by the Sequential I/O procedures, it is opened for the duration of the message printing then closed.

→

```
<operation> = SET^OPENERSPID                        (file must be open)
<new value> = @<openers pid>                                (optional)
<old value> := <openers pid> FOR 4 words                    (optional)
```

Sets allowable openers <process id> for $RECEIVE file. This is used to restrict the openers of this process to a specified process. A typical example is using the Sequential I/O procedures to read the startup message.

Note:
If "open message" = 1 is specified to SET^SYSTEMMESSAGES, the setting of SET^OPENERSPID has no meaning.

```
<operation> = SET^PRINT^ERR^MSG                     (file must be open)
<new value> = <new state>                                   (optional)
<old value> := <state>                                      (optional)
```

Sets/clears print error message for the file. If on and a fatal error occurs, an error message will be displayed on the error file. This is the home terminal unless otherwise specified.

```
<operation> = SET^PROMPT                            (file must be open)
<new value> = <new prompt char>                             (optional)
<old value> := <prompt char>                                (optional)
```

Sets interactive prompt for the file. See the OPEN^FILE procedure.

```
<operation> = SET^READ^TRIM                         (file must be open)
<new value> = <new state>                                   (optional)
<old value> := <state>                                      (optional)
```

Sets/clears read trailing blank trim for the file. If on, the <count read> parameter does not account for trailing blanks.

```
<operation> = SET^RCVEOF                            (file must be open)
<new value> = <new state>                                   (optional)
<old value> := <state>                                      (optional)
```

Sets return EOF on process close for $RECEIVE file. This causes an end-of-file indication to be returned from READ^FILE when the receive open count goes from one to zero; the last close message is received.

The setting for return EOF has no meaning if the user is monitoring open and close messages.

If the file is opened with read-only access, the setting defaults to on for return EOF.

→

```
<operation> = SET^RCVOPENCNT                          (file must be open)
<new value> = <new receive open count>                          (optional)
<old value> := <receive open count>                             (optional)
```

Sets receive-open-count for the $RECEIVE file. This operation is intended to clear the count of openers when an open already accepted by the Sequential I/O procedures is subsequently rejected by the user. See "SET^RCVUSEROPENREPLY".

```
<operation> = SET^RCVUSEROPENREPLY                    (file must be open)
<new value> = <new state>                                       (optional)
<old value> := <state>                                          (optional)
```

Sets user-will-reply for the $RECEIVE file. This is used if the Sequential I/O procedures are to maintain the opener's directory and, therefore, limit opens to a single process[-pair] while keeping the option of rejecting opens.

If <state> is one, <error> of six will be returned from a call to READ^FILE when an open message is received and is the only current open by a process[-pair]. If an open is attempted by a process and an open is currently in effect, the open attempt will be rejected by the Sequential I/O procedures; no return will be made from READ^FILE due to the rejected open attempt.

If <state> is zero, a return from READ^FILE is made only when data is received.

If "open message" = 1 is specified to SET^SYSTEMMESSAGES, the setting of SET^RCVUSEROPENREPLY has no meaning.

<error> of six will be returned from READ^FILE if an open message is accepted by the Sequential I/O procedures.

→

```
<operation> = SET^SYSTEMMESSAGES                    (file must be open)
<new value> = <new sys-msg mask>                                (optional)
<old value> := <sys-msg mask>                                   (optional)
```

Sets system message reception for the $RECEIVE file. Setting a bit in the <sys-msg mask> indicates that the corresponding message is to be passed back to the user. It defaults to the Sequential I/O procedures handling all system messages.

where

```
   <sys-msg-mask>.<0> = break message

                  .<1> = unused
                  .<2> = cpu down
                  .<3> = cpu up

                  .<4> = unused
                  .<5> = stop message
                  .<6> = abend message

                  .<7> = unused
                  .<8> = monitornet
                  .<9> = unused

                 .<10> = open message
                 .<11> = close message
                 .<12> = control message

                 .<13> = setmode message
                 .<14> = resetsync message
                 .<15> = unused
```

The user may reply to the system messages designated by this operation with WRITE^FILE. Note that a reply of <reply error code> of zero is made automatically, if necessary, when a call is made to READ^FILE for the $RECEIVE file.

```
<operation> = SET^TRACEBACK
<new value> = <new state>
<old value> := <old state>
```

Sets/clears the traceback feature. When traceback is active, SIO appends the caller's P-relative address to all error messages.

→

```
<operation> = SET^USERFLAG
<new value> = <new user flag>                          (optional)
<old value> := <user flag in FCB>                      (optional)
```

Sets user flag for the file. The user flag is a one-word value in the FCB which can be manipulated by the user to maintain information about calls to that file.

```
<operation> = SET^WRITE^FOLD                  (file must be open)
<new value> = <new state>                              (optional)
<old value> := <state>                                 (optional)
```

Sets/clears write-fold for the file. If on, writes exceeding the record length cause multiple logical records to be written. If off, writes exceeding the record length are truncated to record length bytes; no error message or warning is given.

```
<operation> = SET^WRITE^PAD                   (file must be open)
<new value> = <new state>                              (optional)
<old value> := <state>                                 (optional)
```

Sets/clears write-blank pad for the file. If on, writes of less than record length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.

```
<operation> = SET^WRITE^TRIM                  (file must be open)
<new value> = <new state>                              (optional)
<old value> := <state>                                 (optional)
```

Sets/clears write-trailing-blank trim for the file. If on, trailing blanks are trimmed from the output record before being written to the file.

Note:
The following operations are used only if the user is performing NOWAIT I/O and calls the AWAITIO procedure directly. See GUARDIAN Programming Manual, File Management section.

```
<operation> = SET^COUNTXFERRED                    (file must be open)
<operation> = <new count>                                  (optional)
<operation> := <count>                                     (optional)
```

Sets the physical I/O count (in bytes) transferred for the file. This is used only if NOWAIT I/O is in effect and the user is making the call to AWAITIO for the file. This is the <count transferred> parameter value returned from AWAITIO.

```
<operation> = SET^ERROR                           (file must be open)
<new value> = <new error>                                  (optional)
<old value> := <error>                                     (optional)
```

Sets file system error code value for the file. This is used only if NOWAIT I/O is in effect and the user makes the call to AWAITIO for the file. This is the <error> parameter value returned from FILEINFO.

```
<operation> = SET^PHYSIOOUT                       (file must be open)
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)
```

Set/clear physical I/O outstanding for file is <file FCB>. This is used only if NOWAIT I/O is in effect and the user is making the call to AWAITIO for the file.

Note:
The following is used only if the user is handling BREAK independently of the Sequential I/O Procedures or if the user has requested break system messages via SET^SYSTEMMESSAGES.

```
<operation> = SET^BREAKHIT
<new value> = <new state>                                  (optional)
<old value> := <state>                                     (optional)
```

Sets/clears break-hit for the file.

The TAKE^BREAK procedure enables break monitoring by a file.

The call to TAKE^BREAK is

```
CALL TAKE^BREAK ( <file FCB> )

where

   <file FCB>, INT:ref,

     identifies the file for which break is to be enabled.
     If the file is not a terminal or if break is already
     enabled for this file, the call will be ignored.

example

   CALL TAKE^BREAK ( out^file );
   WHILE NOT ( break := CHECK^BREAK ( out^file ) ) DO
     BEGIN
       .
       .
       CALL WRITE^FILE ( out^file , buffer , count );
     END;

   CALL GIVE^BREAK ( out^file );
```

The WAIT^FILE procedure is used to wait or check for the completion of an outstanding I/O operation.

The call to WAIT^FILE is

```
<error> := WAIT^FILE (     <file FCB>
                       , [ <count read> ]
                       , [ <time limit> ] )
```

where

<error>, INT,

If abort-on-error mode is in effect, the only possible values for <error> are

- 0 = no error
- 1 = end-of-file
- 6 = system message, only if user has asked for system messages, by SET^SYSTEMMESSAGES
- 40 = operation timed out, only if <time limit> value is supplied and not -1D
- 111 = operation aborted because of break, if break is enabled
- 532 = operation restarted due to retry

<file FCB>, INT:ref,

identifies the file for which there is an outstanding I/O operation.

<count read>, INT:ref,

(optional) is the count of the number of bytes returned due to the requested read operation. The value returned to the parameter has no meaning when waiting for a write operation to complete.

<time limit>, INT(32):value,

(optional) if present indicates whether the caller waits for completion or checks for completion. If omitted, <time limit> is set to -1D.

→

If <time limit> is not 0D then a wait for completion is indicated. The <time limit> then specifies the maximum time, in .01 second units, that the caller waits for a completion. A <time limit> value of -1D indicates a willingness to wait forever.

If <time limit> is 0D then a check for completion is indicated. WAIT^FILE immediately returns to the caller regardless of whether there is a completion. If no completion occurs, the I/O operation is still outstanding; an <error> 40 and an "operation timed out" message is returned.

If <time limit> is 0D and an <error> 40, there is no completion. Therefore, READ^FILE or WRITE^FILE cannot be called for the file until the operation completes by WAIT^FILE. One method of determining if the operation completes is by the CHECK^FILE operation "FILE LOGIOOUT". See "Checking File Transfer Characteristics".

example 1 - wait for completion.

```
CALL READ^FILE ( in^file , buffer ,,,, 1 );
   .
   .
   .
DO error := WAIT^FILE ( in^file , count )
UNTIL error <> SIOERR^IORESTARTED;
```

example 2 - check for completion.

```
IF NOT CHECK^FILE ( recv^file , FILE^LOGIOOUT ) THEN
   CALL READ^FILE ( recv^file , recv^buf ,,,, 1 );
DO error := WAIT^FILE ( recv^file , recv^count , 0D )
UNTIL error <> SIOERR^IORESTARTED;
```

The WRITE^FILE procedure writes a file sequentially. The file must be open with write or readwrite access.

The call to WRITE^FILE is

```
{ CALL        } WRITE^FILE (     <file FCB>
{ <error> := }
                               ,   <buffer>
                               ,   <write count>
                               , [ <reply error code>   ]
                               , [ <forms control code> ]
                               , [ < no wait>           ] )
```

where

<error>, INT,

is a file system or Sequential I/O error indicating the outcome of the write.

If abort-on-error mode, the default case, is in effect, the only possible values for <error> are:

0 = no error

111 = operation aborted because of break, if break is enabled.

If <no wait> is not zero, the only possible value for <error> is zero, when abort-on-error is in effect.

<file FCB>, INT:ref,

identifies the file to which data is to be written.

<buffer>, INT:ref,

is the data to be written. <buffer> must be located within 'G'[0:32767] the process data area.

<write count>, INT:value,

is the count of the number of bytes of <buffer> to be written. A <write count> value of -1 causes SIO to flush the block buffer associated with the file FCB passed. For edit files, flushing the buffer also updates the edit directory on disk.

→

```
    <reply error code>, INT:value,

      (optional, for $RECEIVE file only) is a file management
      error to be returned to the requesting process by REPLY.
      If omitted, zero is replied.

    <forms control code>, INT:value,

      (optional) indicates a forms control operation to be
      performed prior to executing the actual write when the file
      is a process or a line printer.  <forms control> corresponds
      to <parameter> of the file management CONTROL procedure for
      <operation> equal to one.  No forms control will be performed
      if <forms control> is omitted, is negative one, or if the
      file is not a process or a line printer.

    <no wait>, INT:value,

      (optional) indicates whether to wait in this call for the
      I/O to complete.  If omitted or zero, then wait is indicated.
      If <no wait> is not zero, the I/O must be completed in a call
      to WAIT^FILE.

example

   CALL WRITE^FILE ( out^file , buffer , count );
```

A literal is associated with each of the Sequential I/O procedures errors. These messages apply to coding errors and are considered fatal. The one exception is NOWAIT I/O restarted, error SIOERR^IORESTARTED.

The sequential I/O procedure message numbers, messages, and their associated meanings are:

512 SIOERR^INVALIDPARAM
The SIO procedure contains an invalid parameter. This error applies to all procedures. Correct the parameter in error.

513 SIOERR^MISSINGFILENAME
The SIO procedure is missing a file name. This error is an open error. Specify a file name in the procedure call.

514 SIOERR^DEVNOTSUPPORTED
The SIO procedures do not support the specified device type. This error is an open error. Change the device type.

515 SIOERR^INVALIDACCESS
The access mode is not compatible with the device type. This error occurs if the program opens the edit file with read or write access or with blocking specified. This error is an open error. Change either the device type or the access mode.

516 SIO^INVALIDBUFADDR
The buffer address is not within 'G'[0:32767] of the data area. This error is an open error. Move the buffer into lower memory.

517 SIOERR^INVALIDFILECODE
The file code specified in an ASSIGN command does not match the file code of the file. Change either the file name or the file code in the ASSIGN command.

518 SIOERR^BUFTOOSMALL
The buffer specified is too small. For reading an edit file, allocate at least 144 bytes of buffer space. For writing an edit file, allocate at least 1024 bytes of buffer space. For blocking, allocate at least the same number of bytes for buffer space as for the logical record length. If this error persists after increasing the buffer space, the directory of the edit file is in error. Edit the file; the editor usually can correct the directory error. This error is an open error.

519 SIOERR^INVALIDBLKLENGTH
The ASSIGN block length is greater than the block buffer length. Either correct the ASSIGN command or use a larger buffer.

520 SIOERR^INVALIDRECLENGTH
This error occurs when the specified record length is either zero or greater than <max record length> specified in the OPEN^FILE procedure, when the record length for the $RECEIVE file is less than 14, or when the record length is greater than 254 and the procedure specifies variable length records. This error is an open error. Correct the specified record length.

521 SIOERR^INVALIDEDITFILE
An edit file is invalid. This error is an open error.

522 SIOERR^FILEALREADYOPEN
The program used the SET^FILE procedure for a file that should be closed or used the OPEN^FILE procedure for a file that is already open.

523 SIOERR^EDITREADERR
Indicates an edit read error. This error is an open or read error.

524 SIOERR^FILENOTOPEN
The specified file is not open. This error is a check, read, set, write, or wait error. Either open the file or correct the procedure call (for example, change the parameters to permit the operation when the file is closed).

525 SIOERR^ACCESSVIOLATION
The access mode is not compatible with the requested operation. This error is a read or write error. Change either the operation or the access mode.

526 SIOERR^NOSTACKSPACE
The program requires a temporary buffer, but the stack has insufficient space. Increase the run-time memory size if it is less than 32K; otherwise, move one or more non-string arrays to upper memory.

527 SIOERR^BLOCKINGREQD
The program is attempting a write fold or write pad without a block buffer. This error is a write error. Supply a block buffer.

528 SIOERR^EDITDIROVERFLOW
An overflow occurred in the internal directory of an edit file. This error is a write error.

529 SIOERR^INVALIDEDITWRITE
The program attempted to write to an edit file after writing the internal directory. This error is a write error.

530 SIOERR^INVALIDRECVWRITE
The program read the $RECEIVE file, but did not follow the read with a write to the $RECEIVE file. This error is a write error. Either add the missing write or delete the extra read.

531 SIOERR^CANTOPENRECV
The SIO procedure cannot open $RECEIVE for break monitoring. The user did not open the $RECEIVE file with the SIO procedure OPEN^FILE. This error is a CHECK^BREAK error. Open the $RECEIVE file with the OPEN^FILE procedure to do break monitoring while using $RECEIVE.

532 SIOERR^IORESTARTED
The NOWAIT I/O restarted. This message is a warning, not an error. Call WAIT^FILE again to continue waiting.

533 SIOERR^INTERNAL
Indicates there is an internal error. This error is a wait error.

534 SIOERR^CHECKSUMCOM
While performing a checksum on the common FCB, the SIO procedure encountered an error. This error applies to all procedures. Check the program for pointer errors.

535 SIOERR^CHECKSUM
While performing a checksum on the file FCB, the SIO procedure encountered an error. This error applies to all procedures. Check the program for pointer errors.

File characteristics and procedure call information is kept in a File Control Block (FCB) within the user's data space. An FCB is associated with the opening of a file and is passed to each Sequential I/O procedure to identify that file. Additionally, there is one common FCB for each process located within the user's data space. The common FBC contains information common to all files, such as a pointer to the error reporting file.

The common FCB is initialized during the first call to OPEN^FILE following process creation. This is indicated to OPEN^FILE when the first word of the common FCB is set to zero prior to calling OPEN^FILE for the first time.

An FCB is initialized prior to calling OPEN^FILE by invoking the define INIT^FILEFCB or by declaring the FCB using the define ALLOCATE^FCB. The name of the file to be opened must also be put into the FCB by the define ASSIGN^FILENAME.

The FCBs can be located anywhere within the user's data space. The common and file FCBs are linked together forwards and backwards as shown in Figure A-1.

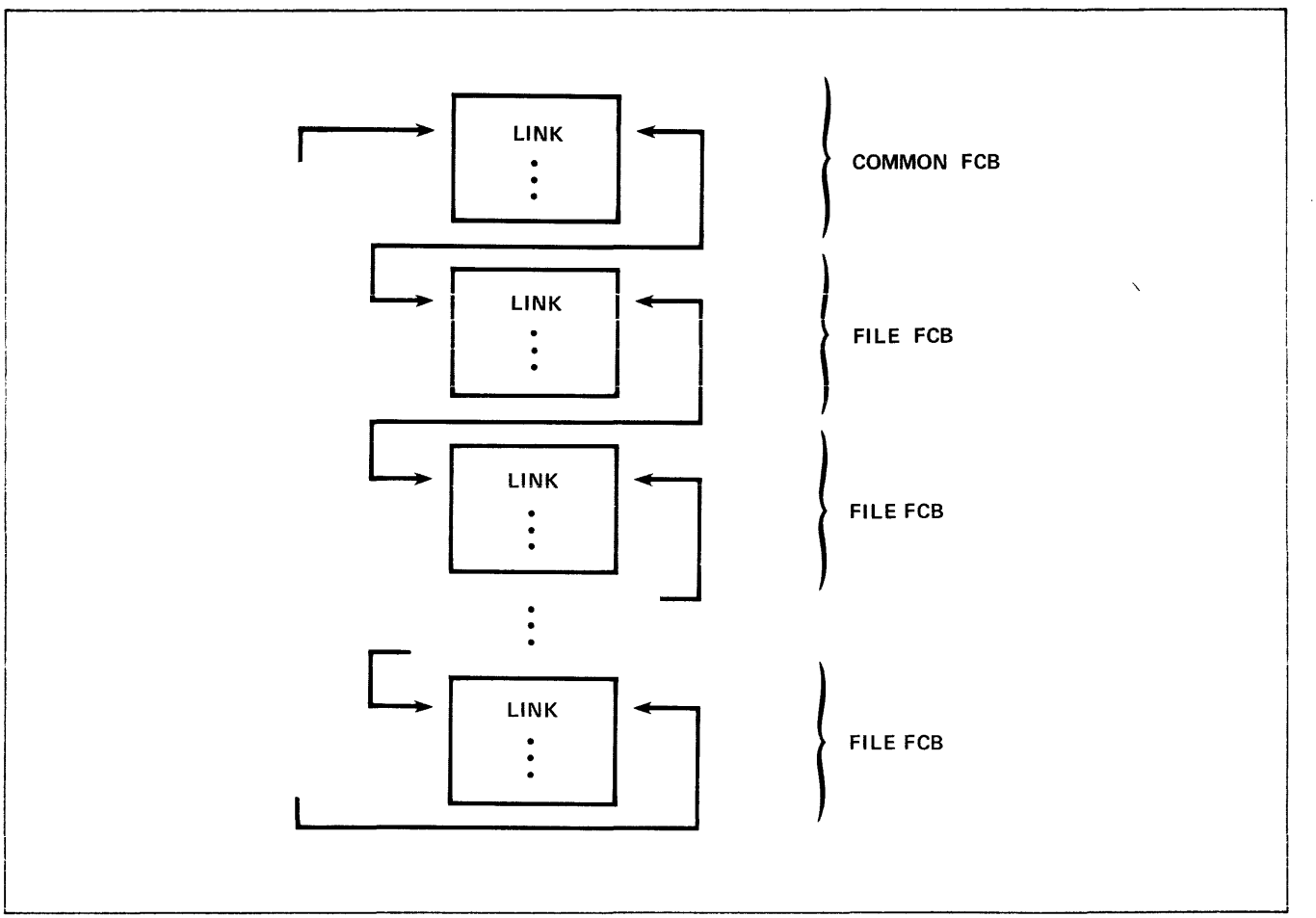


Figure A-1. FCB Linking

The <file FCB> must be allocated and initialized before the OPEN^FILE procedure is called to open a file. The SET^FILE procedure provides these facilities as explained in the following items.

The first three items listed; FCBSIZE, INIT^FILEFCB, and ASSIGN^FILENAME are not used when the INITIALIZER procedure is called to initialize the file control blocks. See the INITIALIZER procedure.

1. The size in words of an FCB is provided as a literal.

   FCBSIZE currently (60)

   example

```
INT .infile [ 0:FCBSIZE-1 ];
```

2. Initialize the FCB using the SET^FILE procedure. This step is required.

   CALL SET^FILE (<file FCB> , INIT^FILEFCB)

   example

```
CALL SET^FILE ( infile , INIT^FILEFCB )
```

3. Specify the name of the file to open. This step is required.

   CALL SET^FILE (<file FCB> , ASSIGN^FILENAME , <file name addr>)

   example

```
CALL SET^FILE ( in^file, ASSIGN^FILENAME , @in^filename );
```

4. Specify the access mode for this open. This step is optional.

   CALL SET^FILE (<file FCB> , ASSIGN^OPENACCESS , <open access>)

   The following LITERALS are provided for <open access> :

```
READWRITE^ACCESS (0)
READ^ACCESS      (1)
WRITE^ACCESS     (2)
```

If omitted, the access mode defaults for the device being opened to the following:

Device	Access
Process	Read/Write
Operator	Write
$RECEIVE	Read/Write
Disc	Read/Write
Mag Tape	Read/Write
Printer	Write
Terminal	Read/Write
Card Reader	Read

example

```
CALL SET^FILE ( in^file , ASSIGN^OPENACCESS , READ^ACCESS );
```

5. Specify exclusion for this open. This step is optional.

```
CALL SET^FILE ( <file FCB> , ASSIGN^OPENEXCLUSION ,
                <open exclusion> )
```

The following LITERALS are provided for <open exclusion> :

```
SHARE     (0)
EXCLUSIVE (1)
PROTECTED (3)
```

If omitted, the exclusion mode applied to the open, defaults to the following:

Access	Exclusion Mode
Read	if terminal then share else protected
Write	if terminal then share else exclusive
Read/Write	if terminal then share else exclusive

example

```
CALL SET^FILE ( in^file , ASSIGN^OPENEXCLUSION , EXCLUSIVE );
```

6. Specify the logical record length. This step is optional.

```
CALL SET^FILE ( <file FCB> , ASSIGN^RECORDLENGTH ,
                <record length> )
```

The <record length> is given in bytes.

If omitted, <record length> defaults according to the device as follows:

Device	Logical Record Length
Operator	132 bytes
Process	132 bytes
$RECEIVE	132 bytes
unstructured disc	132 bytes
structured disc	record length defined at creation
mag tape	132 bytes
terminal	132 bytes
printer	132 bytes
card reader	132 bytes

7. Set the file code. This step is optional and has two meanings: 1) if AUTO^CREATE is on, the file code specifies the type of file to be created. 2) implies the file code must match the file code specified for the open to succeed.

   CALL SET^FILE (<file FCB> , ASSIGN^FILECODE , <file code>)

8. Set the primary extent size. This step is optional and has meaning only if AUTO^CREATE is on.

   CALL SET^FILE (<file FCB> , ASSIGN^PRIMARYEXTENTSIZE ,
   <primary extent size>)

   <primary extent size> is given in pages (2048-byte units).

9. Set the secondary extent size. This step is optional and has meaning only if AUTO^CREATE is on.

   CALL SET^FILE (<file FCB> , ASSIGN^SECONDARYEXTENTSIZE ,
   <secondary extent size>)

   <secondary extent size> is given in pages, 2048-byte units.

10. Set the file's physical block length. This step is optional. It is the number of bytes transferred between the file and the process in a single I/O operation. If the <block length> is specified, blocking is also specified. A physical block is composed of <block length> divided by <record length> logical records. When <block length> is not exactly divisible by <record length>, the portion of that block following the last logical record is filled with blanks.

Note that the specified form of blocking differs from the type of blocking performed when no <block length> is specified. In the unspecified form, there is no indication of a physical block size, the records are contiguous on the media.

CALL SET^FILE (<file FCB> , ASSIGN^BLOCKLENGTH , <block length>)

<block length> is given in bytes.

The Sequential I/O procedures and the INITIALIZER procedure can be used in conjunction with or separately from each other. File transfer characteristics, such as record length, can be altered at run time using the Command Interpreter ASSIGN commands. See the INITIALIZER procedure, in the GUARDIAN Programming Manual.

The INITIALIZER is a GPLIB procedure that reads the startup and, optionally, the ASSIGN and PARAM messages. The INITIALIZER procedure can prepare global tables of a predefined structure and properly initialize FCBs with the information read from the startup and ASSIGN messages.

To use the INITIALIZER, an array called a Run-Unit Control Block must be declared. Each file to be prepared by the INITIALIZER must be initialized with a default physical file name and, optionally, with a logical file name before invoking the INITIALIZER.

The INITIALIZER reads the startup message then requests the ASSIGN messages. For each ASSIGN message, the FCBs are searched for a logical file name which matches the logical file name contained in the ASSIGN message. If a match is found, the information from the ASSIGN message is put into the FCB. See the GUARDIAN Programming Manual, Application Interface section, or the GUARDIAN Command Language and Utilities Manual, Comint section, for a description of the ASSIGN Command.

The INITIALIZER also substitutes the real file names for default physical file names into the FCBs. This function provides the capability to define the IN and OUT files of the startup message as physical files and to define the home terminal as a physical file.

After invoking the INITIALIZER, the Sequential I/O OPEN^FILE procedure is called once for each file to be opened.

INITIALIZER-RELATED DEFINES

Two Defines are provided for allocating Run-Unit Control Block Space (CBS) and for allocating FCB space. These defines are:

1. Allocate Run-Unit Control Block and Common FCB (data declaration).

```
ALLOCATE^CBS ( <run-unit control block> , <common FCB> ,
               <numfiles> );
```

where

<run-unit control block>

is the name to be given to the run-unit control block, this name is passed to the INITIALIZER procedure.

<common FCB>

is the name to be given to the common FCB, this name is passed to the OPEN^FILE procedure.

<numfiles>

is the number of FCBs to be prepared by the INITIALIZER procedure. The INITIALIZER begins with the first FCB following ALLOCATE^CBS.

example

```
ALLOCATE^CBS ( rucb , commfcb , 2 );
```

2. Allocate FCB (data declaration).

Note: The FCB allocation Defines must immediately follow the ALLOCATE^CBS Define. No intervening variables are allowed.

ALLOCATE^FCB (<file FCB> , <default physical file name>)

where

<file FCB>

is the name to be given the FCB. The name references the file in other Sequential I/O procedure calls.

<default physical file name>, literal STRING,

is the name of the file to be opened. This can be an internal form of a file name or one of the following and must be in upper case as shown.

byte numbers

[0] [8] [16] [24]

" #IN "

This means substitute the INFILE name of the startup message for this name.

" #OUT "

This means substitute the OUTFILE name of the startup message for this name.

" #TERM "

This means substitute the home terminal name for this name.

```
"        #TEMP          "
```

This means substitute a name appropriate for creating a temporary file for this name.

```
"                       "
```

This means substitute a name appropriate for creating a temporary file for this name.

If the $<volume name> or <subvol name> is omitted, the corresponding default name from the startup message is substituted for the disc file names.

example

```
ALLOCATE^FCB ( in^file  , "        #IN             " );
ALLOCATE^FCB ( out^file , "        #OUT            " );
```

The following SET^FILE operation, ASSIGN^LOGICALFILENAME, is used with the INITIALIZER. The logical file name is the means by which the INITIALIZER matches an ASSIGN message to a physical file.

```
CALL SET^FILE ( <file FCB> , ASSIGN^LOGICALFILENAME ,
                @<logical file name> )
```

where

<file FCB>, INT:ref,

references the file to be assigned a logical file name.

@<logical file name>, INT:value,

is the word address of an array containing the logical file name. A logical file name consists of a maximum of seven alphanumeric characters the first of which must be an alpha character.

<logical file name> must be encoded as follows

byte numbers

```
[0]  [1]                [8]
<len><logical file name>
```

<len> is the length of the logical file name.

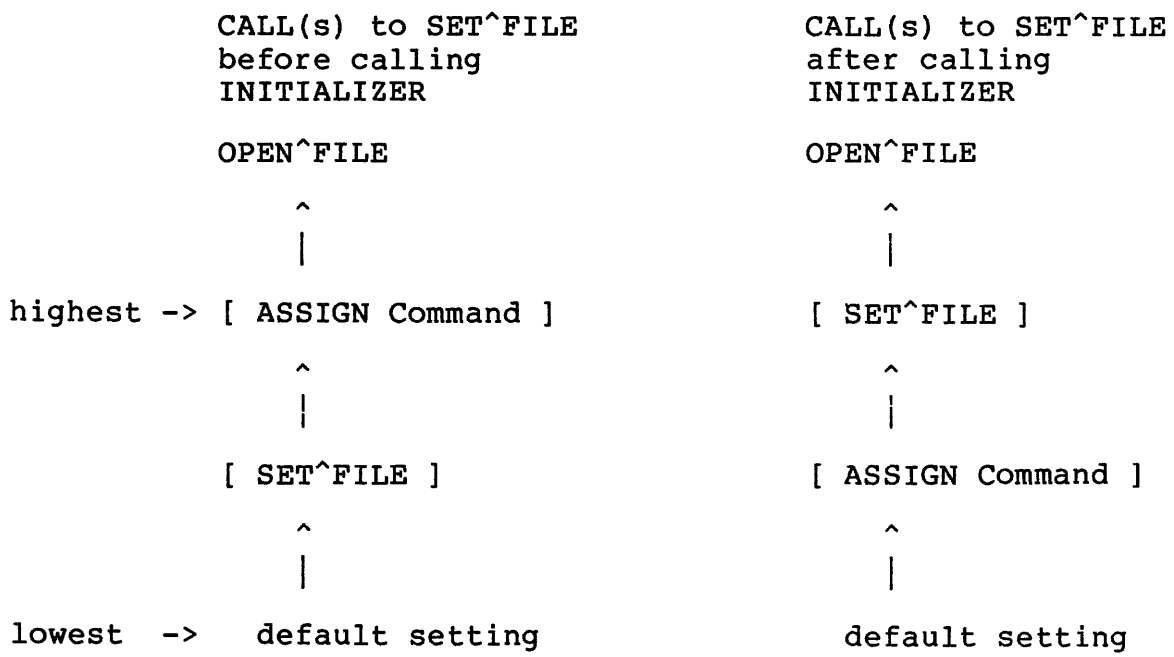
By convention, the logical file name of the input file of the startup message should be named "INPUT"; the logical file name of the output file of the startup message should be named "OUTPUT".

example

```
INT .buf [ 0:11 ];
STRING .sbuf := @buf '<<' 1;

sbuf ':=' [ 5, "INPUT" ];
CALL SET^FILE (  in^file , ASSIGN^LOGICALFILENAME , @buf );
sbuf ':=' [ 6, "OUTPUT" ];
CALL SET^FILE (  out^file , ASSIGN^LOGICALFILENAME , @buf );
```

The following shows the file assignment in relation to when the INITIALIZER is invoked. File characteristics can be set by the INITIALIZER with the ASSIGN command or with programmatic calls to the SET^FILE procedure.

Precedence of Setting File Characteristics



CONSIDERATIONS

- If run-time changes to file transfer characteristics are not allowed then do not assign a logical file name to the file.

- In some cases it is undesirable to have the INITIALIZER assign a physical file name for the <default physical file name>. For example, when it is not desirable to default the file name, but instead to force the use of an ASSIGN command to specify a physical file for the logical file, then declare the FCB as follows (the FCB must be adjacent to other FCBs searched by the INITIALIZER):

```
INT .<file FCB> [ 0:FCBSIZE - 1 ];
```

  In the executable part of the program, before calling the INITIALIZER, initialize the FCB:

```
CALL SET^FILE ( <file FCB> , INIT^FILEFCB );
```

  Assign a logical file name, and any other open attributes desired, before calling the INITIALIZER:

```
CALL SET^FILE ( <file FCB> , ASSIGN^LOGICALFILENAME , @name );
   .
CALL INITIALIZER ( .. );

CALL OPEN^FILE ( <common FCB> , <file FCB> , ... );
```

  If the user neglects to ASSIGN a physical file to the logical file, the open will fail with an error number 513, SIOERR^MISSINGFILENAME, file name is not supplied.

USAGE EXAMPLE

The following shows the use of the INITIALIZER and Sequential I/O procedures for opening the IN and OUT files of a typical TANDEM subsystem program.

If the IN and OUT files are the same file and either is a terminal or a process, only the IN file is opened. The address of the in^file FCB is put into the pointer to the out^file FCB.

The open access is assigned after the INITIALIZER is called. This overrides the open access specified in an ASSIGN command.

```
?NOLIST, SOURCE $SYSTEM.SYSTEM.GPLDEFS
?LIST
! Set up the control blocks for the initializer with supplied
!DEFINEs.
! Initialize Run Unit Control Block and common FCB.
!     rucb     - Array holding Run Unit Control Block.
!     commfcb  - Array for the common File Control Block.

    ALLOCATE^CBS ( rucb, commfcb, 2 );
```

SIO: USAGE EXAMPLE WITH THE INITIALIZER

```
! Initialize in file FCB.
!     in^file  - Array for FCB of the in file.

    ALLOCATE^FCB ( in^file,  "        #IN             " );

! Initialize out file FCB.
!     out^file - Array for FCB of the out file.

    ALLOCATE^FCB ( out^file, "        #OUT            " );

LITERAL
     process   =    0,               ! Process device type.
     terminal  =    6,               ! Terminal device type.
     inblklen  = 4096,               ! Length of block buffer for in
                                     ! file.
     outblklen = 4096,               ! Length of block buffer for out
                                     ! file.
     rec^len   =  255;               ! Maximum record length to read
                                     ! or write.

INT .inblkbuf [ 0:inblklen/2 - 1 ],  ! In file's buffer for blocking.
    .outblkbuf [ 0:outblklen/2 - 1 ],! Out file's buffer for blocking.
    .infname,                        ! In file's file name.
    .outfname,                       ! Out file's file name.
     device^type,                    ! Device type, GUARDIAN manual,
                                     ! p2.3-21.
     phys^rec^len,                   ! Physical record length of
                                     ! device.
     interactive;                    ! Indicates if in and out file
                                     ! are interactive implying use
                                     ! READWRITE access.

INT .buf [ 0:11 ];                   ! Holds logical file names.
STRING
    .sbuf := @buf '<<' 1;            ! String corresponding to buf.

?NOLIST, SOURCE $SYSTEM.SYSTEM.EXTDECS
?LIST

PROC main^proc MAIN;
  BEGIN
    int .buffer [ 0:rec^len/2 - 1 ], ! Buffer for i/o with a single
                                     ! record.
        count := rec^len;            ! Number of bytes read in or
                                     ! written out.
```

```
!     Beginning of program execution.

!     Set up in and out files using startup message from RUN command.
        sbuf ':=' [ 5, "INPUT" ];
        CALL SET^FILE( in^file, ASSIGN^LOGICALFILENAME, @buf );
        sbuf ':=' [ 6, "OUTPUT" ];
        CALL SET^FILE( out^file, ASSIGN^LOGICALFILENAME, @buf );
        CALL INITIALIZER( rucb );

!         get physical file names for in and out files.

          @infname  := CHECK^FILE( in^file,  FILE^FILENAME^ADDR );
          @outfname := CHECK^FILE( out^file, FILE^FILENAME^ADDR );

!     Determine type of access for in file.

        CALL DEVICEINFO ( infname, device^type, phys^rec^len );
        interactive :=
          IF ( device^type.<4:9> = terminal  OR
               device^type.<4:9> = process )
             AND infname = outfname FOR 12
             THEN 1 ELSE 0;

        CALL SET^FILE( in^file, ASSIGN^OPENACCESS,
                       IF interactive THEN READWRITE^ACCESS
                  (                   ELSE READ^ACCESS );

!     Open in file.

        CALL OPEN^FILE( commfcb, in^file, inblkbuf
                        ,inblklen,,,,, out^file );

    IF interactive THEN       ! Make in and out files the same,
                              ! no need to
      @out^file := @in^file   ! open out file.
    ELSE                      ! Open out file.
      BEGIN
        CALL SET^FILE( out^file, ASSIGN^OPENACCESS, WRITE^ACCESS );
        CALL OPEN^FILE( commfcb, out^file, outblkbuf, outblklen );

!         non-interactive use so echo reads to out file.

        CALL SET^FILE( in^file, SET^DUPFILE, @out^file );
      END;
```

```
!     Main processing loop.

!     WHILE not EOF process the record.

    WHILE ( READ^FILE( in^file, buffer, count) ) <> 1 DO
      BEGIN

!
!         Process record read in and format a record for output.
!           .
!           .

        CALL WRITE^FILE( out^file, buffer, count );
      END;

   CALL CLOSE^FILE( commfcb );     ! close all files

  END;                             ! of main^proc
```

To change the record length of the input file, the following ASSIGN command can be entered before the program is run.

```
ASSIGN INPUT,,REC 80
```

To change the file code of the output file, the following ASSIGN command can be entered before the program is run.

```
ASSIGN OUTPUT,,CODE 9876
```

SUMMARY

The following are the steps involved to use the INITIALIZER with the Sequential I/O procedures.

- Allocate the CBS and FCB and assign the default physical file names using ALLOCATE^CBS and ALLOCATE^FCBs.
- Assign a logical file name using the SET^FILE operation, ASSIGN^LOGICALFILENAME.
- If ASSIGN command characteristics are to override program calls to SET^FILE, invoke assignment defines.
- Invoke the INITIALIZER to read the startup, ASSIGN, and PARAM messages and prepare the file FCBs.
- If programmatic calls to SET^FILE are to override ASSIGN command characteristics, invoke assignment defines.
- Open the files with calls to OPEN^FILE.

The following example shows the use of the Sequential I/O procedures for the IN and OUT files of a typical TANDEM subsystem program when the INITIALIZER procedure is not used.

```
?SOURCE $SYSTEM.SYSTEM.GPLDEFS ( ... )
INT  interactive,
     error,
    .common^fcb [0:FCBSIZE-1] := 0,
    .rcv^file   [0:FCBSIZE-1],
    .in^file    [0:FCBSIZE-1],
    .out^file   [0:FCBSIZE-1],
    .buffer     [0:99],
     mompid     [0:3],
     devtype,
     junk;

LITERAL
     process       =     0,
     terminal      =     6,
     in^blkbuflen  =  1024,
     out^blkbuflen =  1024;

INT .in^blkbuf  [0:in^blkbuflen/2  - 1],
    .out^blkbuf [0:out^blkbuflen/2 - 1];

?SOURCE $SYSTEM.SYSTEM.EXTDECS ( ... )
!
! read the startup message.
!
! - open $receive.
!
CALL SET^FILE ( rcv^file , INIT^FILEFCB );
buffer ':=' "$RECEIVE  " & buffer [ 4 ] FOR 7;
! file name.
CALL SET^FILE ( rcv^file , ASSIGN^FILENAME , @buffer );
! number of bytes to read.
CALL SET^FILE ( rcv^file , ASSIGN^RECORDLENGTH , 200 );
CALL OPEN^FILE ( common^fcb , rcv^file ,,, nowait , nowait );
!
! - get moms process id.
!
! - first, see if i'm named.
!
CALL GETCRTPID ( MYPID , buffer );
IF buffer.<0:1> = 2 THEN
   ! not named.
   CALL MOM ( mompid );
ELSE
  BEGIN
    ! named.
    CALL LOOKUPPROCESSNAME ( buffer );
    mompid ':=' buffer [ 5 ] FOR 4;
  END;
! - allow startup message from mom only.
```

```
CALL SET^FILE ( rcv^file , SET^OPENERSPID , @mompid );
!
DO
  BEGIN
    CALL READ^FILE ( rcv^file , buffer ,,,, 1 );
    DO error := WAIT^FILE ( rcv^file , length , 3000D )
    UNTIL error <> SIOERR^IORESTARTED;
  END
UNTIL buffer = -1; ! startup message read.

! - close $receive.
CALL CLOSE^FILE ( rcv^file );
!
! see if program is being run interactively.
!
CALL DEVICEINFO ( buffer [ 9 ] , devtype , junk );
interactive :=
   IF ( devtype.<4:9> = terminal OR
        devtype.<4:9> = process  ) AND
        buffer [ 9 ] = buffer [ 21 ] for 12 THEN 1
                                            ELSE 0;

CALL SET^FILE ( in^file , INIT^FILEFCB );
CALL SET^FILE ( in^file , ASSIGN^FILENAME , @buffer [ 9 ] );
CALL SET^FILE ( in^file , ASSIGN^OPENACCESS ,
                IF interactive THEN READWRITE^ACCESS
                               ELSE READ^ACCESS );
CALL OPEN^FILE ( common^fcb , in^file , in^blkbuf , in^blkbuflen
                 ,,,,, out^file );

IF interactive THEN
   ! use in file as out file.
   @out^file := @in^file
ELSE
  BEGIN
    CALL SET^FILE ( out^file , INIT^FILEFCB );
    CALL SET^FILE ( out^file , ASSIGN^FILENAME , @buffer [ 21 ] );
    CALL SET^FILE ( out^file , ASSIGN^OPENACCESS , WRITE^ACCESS );
    CALL OPEN^FILE ( common^fcb , out^file , out^blkbuf ,
                     out^blkbuflen );
    ! set duplicative file.
    CALL SET^FILE ( in^file , SET^DUPFILE , @out^file );
  END;
.
.
.
```

Error handling and retries are implemented within the Sequential I/O procedure environment by the NO^ERROR procedure. NO^ERROR is called internally by the Sequential I/O procedures. If the file is opened by OPEN^FILE then the NO^ERROR procedure can be called directly for the file system procedures.

The call to NO^ERROR is

```
{ <no retry> := }
{ CALL          } NO^ERROR (   <state>
                             , <file FCB>
                             , <good error list>
                             , <retryable> )
```

where

<no retry>, INT,

indicates whether or not the I/O operation should be retried. Values of <no retry> are:

0 = operation should be retried.
<>0 = operation should not be retried.

If <no retry> is not zero, one of the following is indicated:

- <state> is not zero.
- no error occurred, error is zero.
- error is a good error number on the list.
- fatal error occurred and abort-on-error mode is off.
- error is a break error and break is enabled for <file FCB>.

<state>, INT:value,

if non-zero, indicates the operation is to be considered successful. The file error and retry count variables are set to zero with <no retry> returned as non-zero. Typically, either of two values is passed in this position:

= CCE for example, immediately following a file system call. If equal is true, the operation is successful. This eliminates a call to FILEINFO by NO^ERROR.

0 forces NO^ERROR to first check the error value in the FCB. If the FCB error is zero, NO^ERROR calls FILEINFO for the file.

→

```
  <file FCB>, INT:ref,

      identifies the file to be checked.

    <good error list>, INT:ref,

      is a list of error numbers; if one of the numbers matches
      the current error, <no retry> is returned as non-zero
      (no retry).  The format of <good error list>, in words, is

         word [ 0 ] = number of error numbers in list {0:n}
         word [ 1 ] = good error number
              .
              .
              .
         word [ n ] = good error number.

    <retryable>, INT:value,

      is used to determine whether certain path errors should be
      retried.  If <retryable> is not zero, errors in the
      range of {120, 190, 202:231} will be retried according
      to the device type as follows:

         Device          Retry Indication

         Process         n.a.
         Operator        yes
         $RECEIVE        n.a.
         Disc            (opened with sync depth of 1 so n.a.)
         Mag Tape        no
         Printer         yes
         Terminal        yes
         Card Reader     no

      If the path error is either of {200:201} a retry indication
      will be given in all cases following the first attempt.

example

    INT good^error [ 0:1 ] := [ 1, 11 ]; ! nonexistent record.

    CALL SET^FILE ( out^file, SET^ERROR, 0)
    DO CALL READUPDATE ( out^fnum, buffer , count )
    UNTIL NO^ERROR ( = , out^file , good^error , 0 );
```

ERROR HANDLING BY NO^ERROR

Errors are handled as follows:

```
if <state> then
  begin
    fcb^error := 0;
    retrycount := 0;
    return no-retry indication
  end;

if not fcb^error then
   CALL FILEINFO ( fcb^fnum , fcb^error );
```

fcb^error	Disposition
0	return no-retry indication
1,6	READ^FILE: return no-retry indication
7	WRITE^FILE: return no-retry indication
<good^error>	return no-retry indication
100:102	prompt then if "S[TOP]" then fatal else return retry indication

110:111	if device = breakdevice then
	begin
	breakflush := 1;
	if (breakhit :=
	checkbreak
	begin
	check $receive for break message.
	if break message then
	breaktyped := 1
	else
	if breakflush then
	begin
	take break
	delay 2 sec
	end
	return breaktyped.
	end) then return no-retry indication.
	end
	delay 2 sec
	return retry indication
	end

112	begin delay 2 sec return retry indication end

```
200:201          if ( retrycount := retrycount + 1 ) > 1 then
                    goto fatal
                 else return retry indication.

120, 190         if not retryable or
202:231             (retrycount := retrycount + 1) > 1 then
                    goto fatal
                 else
                 if device <> mag tape    and
                    device <> card reader then
                       return retry indication
                 else
                    goto fatal

other            fatal:
                   if print error then
                      print an error message;
                   if abort then
                     begin
                       call close^file ( common^fcb );
                       call abend;
                     end;
                   return no-retry indication;
```

The retry count is used to determine the number of times an operation is consecutively retried for a maximum of two retries. The count is cleared when a no-retry is indicated.

Within the environment of the Sequential I/O procedures, the $RECEIVE file has two functions.

. To check for break messages.

. To transfer data between processes.

Within the Sequential I/O procedures, these functions can be performed concurrently. It may be desirable to manage the $RECEIVE file independently of the Sequential I/O procedures, and to monitor break using the Sequential I/O procedures. Therefore, the SET^FILE operation SET^BREAKHIT enables the user's $RECEIVE handler to pass the break information into the Sequential I/O procedure environment.

$RECEIVE DATA TRANSFER PROTOCOL

```
RS  = RECEIVE^STATE: 0 = NEED READUPDATE, 1 = NEED REPLY.
ROC = RECEIVE^OPENER^COUNT.

  OPEN^FILE
     RS := ROC := 0;

  READ^FILE (file must be open with read or readwrite access)

     if system message then
       begin
         RS := 1
         if user wants to process this message then
            return 1;
         replycode := 0
         if cpu down message then
           begin
             if cpu = opener's cpu then
                { delete process from opener's directory }
           end
         else
         if break^message then
           begin
             breakhit := 1
           end
         else
         if open^message then
           begin
             if nowait depth > 1 then replycode := 2
             else
             if ROC = 2 then
                replycode := 12
             else
```

```
        if primary open then
          begin
            if not primary pid or
               opener = primary pid then
              begin
                add primary pid to opener directory
                ROC := ROC + 1
              end
            else replycode := 12
          end
        else
        if backup open and
           ( pid in message = primary openers pid or
             not primary pid ) then
          begin
            if primary pid then
              add backup pid to opener directory
            else
              ! treat as primary open.
              add primary pid to opener directory
            ROC := ROC + 1
          end
        else replycode := 12
      end
    else
    if close message then
      begin
        if pid = primary pid then
          begin
            primary pid := backup pid
            delete backup pid from opener directory
          end
        else
        if pid = backup pid then
           delete backup pid from opener directory
        if not ( ROC := ROC - 1 ) and
           rcveof then
           error := 1
      end.

    if open message and
       user wants to reply
       and not replycode then return 1
    else
      begin
        REPLY ( replycode )
        RS := 0
      end

    return if  error = 1 then 0 else 1
  end ) then return.
```

```
   if RS then REPLY ( no text, REPLYERROR = 0 ); RS := 0;
      Note: REPLY is skipped if READ^FILE immediately follows
            open.
   READUPDATE ( text ); RS := 1;
   error := 0;
```

WRITE^FILE (file must be open with write or writeread access)

```
   if not RS then ! invalid operation
      error := SIOERR^INVALIDRCVWRITE
      RETURN;
   REPLY ( text, reply code ); RS := 0;
   error := 0;
```

CLOSE^FILE

```
   replycode := IF access = write THEN 1 ELSE 45;
   if not RS then READUPDATE ( no text ); RS := 1;
   REPLY ( no text, replycode ); RS := 0;

   READUPDATE/REPLY until close message; RS := 0;
```

Note: To determine whether the data returned from READ^FILE is listing text or command prompt text call the file system RECEIVEINFO procedure.

If NOWAIT is specified at open time, the file is opened with a NOWAIT I/O depth of one. Whether an individual operation is to be waited for is determined on a call by call basis. NOWAIT operations are completed by a call to WAIT^FILE.

If it is desirable to wait for any file, the user can call AWAITIO before calling WAIT^FILE. Dependent on whether blocking is performed, a physical I/O may not always take place with a logical I/O. Therefore, the CHECK^FILE operation FILE^PHYSIOOUT is used to determine if an I/O is outstanding. The SET^FILE operations SET^PHYSIOOUT, SET^ERROR, and SET^COUNTXFERRED are provided to condition the FCB, if the I/O is completed. The user must call WAIT^FILE following the call to AWAITIO for the file state information to be updated.

example

```
INT .in^fnum;

@in^fnum := CHECK^FILE ( in^file , FILE^FNUM );
error := 0;
WHILE 1 DO
  BEGIN
    IF error <> SIOERR^IORESTARTED THEN
       CALL READ^FILE ( in^file , buffer , , , 1 ); ! no wait.
      .
      .
      .
    fnum := -1;
    CALL AWAITIO ( fnum ,, countread ,, 3000D );
    IF fnum = in^fnum THEN
      BEGIN
        CALL FILEINFO ( in^fnum ( in^file ) , error );
        ! set I/O done.
        CALL SET^FILE ( in^file , SET^PHYSIOOUT , 0 );
        ! set count read.
        CALL SET^FILE ( in^file , SET^COUNTXFERRED , countread );
        ! set error code.
        CALL SET^FILE ( in^file , SET^ERROR , error );
        IF ( error :=
           WAIT^FILE ( in^file , in^file^countread ) ) <>
              SIOERR^IORESTARTED THEN
          BEGIN ! completed.
            !
            ! process read.
            !
          END;
      END
    ELSE
      .
      .
  END; ! WHILE 1 LOOP.
```

The following is the $SYSTEM.SYSTEM.GPLDEFS source file for the Sequential I/O procedures.

```
?PAGE "T9600D00 - SIO PROCEDURES - DEFINITIONS"
!
! FCB SIZE IN WORDS.
!
LITERAL
     FCBSIZE = 60;
!
! DECLARE RUCB , PUCB, AND COMMON FCB.
!
DEFINE
     ALLOCATE^CBS ( RUCB^NAME , COMMON^FCB^NAME , NUM^FILES ) =
        INT .RUCB^NAME [ 0:65 ] :=
               ! RUCB PART.
               [ 62 , 1 , 27 * [ 0 ] , 62 , 32 * [ 0 ] ,
               ! PUCB PART.
                 4 , NUM^FILES , 4 + FCBSIZE ];
        INT .COMMON^FCB^NAME [ 0:FCBSIZE - 1 ] :=
               [ FCBSIZE * [ 0 ] ]#;
!
! DECLARE FCB.
!
DEFINE
     ALLOCATE^FCB ( FCB^NAME , PHYS^FILENAME ) =
        INT .FCB^NAME [ 0:FCBSIZE - 1 ] :=
              [ FCBSIZE , %000061 , -1 , %100000 , 0 , PHYS^FILENAME ,
                ( FCBSIZE - 17 ) * [ 0 ] ]#;
!
! SET^FILE OPERATIONS.
!
LITERAL
      INIT^FILEFCB                =   0,
      !
      ASSIGN^FILENAME             =   1,
      ASSIGN^LOGICALFILENAME      =   2,
      ASSIGN^OPENACCESS           =   3,
      ASSIGN^OPENEXCLUSION        =   4,
      ASSIGN^RECORDLENGTH         =   5,
      ASSIGN^FILECODE             =   6,
      ASSIGN^PRIMARYEXTENTSIZE    =   7,
      ASSIGN^SECONDARYEXTENTSIZE  =   8,
      ASSIGN^BLOCKLENGTH          =   9,
      !
      SET^DUPFILE                 =  10,
      SET^SYSTEMMESSAGES          =  11,
      SET^OPENERSPID              =  12,
      SET^RCVUSEROPENREPLY        =  13,
      SET^RCVOPENCNT              =  14,
      SET^RCVEOF                  =  15,
      SET^USERFLAG                =  16,
      SET^ABORT^XFERERR           =  17,
      SET^PRINT^ERR^MSG           =  18,
```

```
    SET^READ^TRIM                = 19,
    SET^WRITE^TRIM               = 20,
    SET^WRITE^FOLD               = 21,
    SET^WRITE^PAD                = 22,
    SET^CRLF^BREAK               = 23,
    SET^PROMPT                   = 24,
    SET^ERRORFILE                = 25,
    SET^PHYSIOOUT                = 26,
    SET^LOGIOOUT                 = 27,
    SET^COUNTXFERRED             = 28,
    SET^ERROR                    = 29,
    SET^BREAKHIT                 = 30,
    SET^TRACEBACK                = 31,
    !
    SET^EDITREAD^REPOSITION      = 32,
    !
    FILE^FILENAME^ADDR           = 33,
    FILE^LOGICALFILENAME^ADDR    = 34,
    FILE^FNUM^ADDR               = 35,
    FILE^ERROR^ADDR              = 36,
    FILE^USERFLAG^ADDR           = 37,
    FILE^SEQNUM^ADDR             = 38,
    FILE^FILEINFO                = 39,
    FILE^CREATED                 = 40,
    FILE^FNUM                    = 41,
    FILE^SEQNUM                  = 42,
    !
    MAX^OPERATION                = 42,
    !
    FILE^FILENAME                = ASSIGN^FILENAME              + 256,
    FILE^LOGICALFILENAME         = ASSIGN^LOGICALFILENAME       + 256,
    FILE^OPENACCESS              = ASSIGN^OPENACCESS            + 256,
    FILE^OPENEXCLUSION           = ASSIGN^OPENEXCLUSION         + 256,
    FILE^RECORDLEN               = ASSIGN^RECORDLENGTH          + 256,
    FILE^FILECODE                = ASSIGN^FILECODE              + 256,
    FILE^PRIEXT                  = ASSIGN^PRIMARYEXTENTSIZE     + 256,
    FILE^SECEXT                  = ASSIGN^SECONDARYEXTENTSIZE   + 256,
    FILE^BLOCKBUFLEN             = ASSIGN^BLOCKLENGTH           + 256,
    FILE^DUPFILE                 = SET^DUPFILE                  + 256,
    FILE^SYSTEMMESSAGES          = SET^SYSTEMMESSAGES           + 256,
    FILE^OPENERSPID              = SET^OPENERSPID               + 256,
    FILE^RCVUSEROPENREPLY        = SET^RCVUSEROPENREPLY         + 256,
    FILE^RCVOPENCNT              = SET^RCVOPENCNT               + 256,
    FILE^RCVEOF                  = SET^RCVEOF                   + 256,
    FILE^USERFLAG                = SET^USERFLAG                 + 256,
    FILE^ABORT^XFERERR           = SET^ABORT^XFERERR            + 256,
    FILE^PRINT^ERR^MSG           = SET^PRINT^ERR^MSG            + 256,
    FILE^READ^TRIM               = SET^READ^TRIM                + 256,
    FILE^WRITE^TRIM              = SET^WRITE^TRIM               + 256,
    FILE^WRITE^FOLD              = SET^WRITE^FOLD               + 256,
    FILE^WRITE^PAD               = SET^WRITE^PAD                + 256,
    FILE^CRLF^BREAK              = SET^CRLF^BREAK               + 256,
    FILE^PROMPT                  = SET^PROMPT                   + 256,
    FILE^ERRORFILE               = SET^ERRORFILE                + 256,
```

```
      FILE^PHYSIOOUT              = SET^PHYSIOOUT              + 256,
      FILE^LOGIOOUT               = SET^LOGIOOUT               + 256,
      FILE^COUNTXFERRED           = SET^COUNTXFERRED           + 256,
      FILE^ERROR                  = SET^ERROR                  + 256,
      FILE^BREAKHIT               = SET^BREAKHIT               + 256,
      FILE^TRACEBACK              = SET^TRACEBACK              + 256;
!
! OPEN ACCESS.
!
LITERAL
     READWRITE^ACCESS = 0,
     READ^ACCESS      = 1,
     WRITE^ACCESS     = 2;
!
! OPEN EXCLUSION.
!
LITERAL
     SHARE     = 0,
     EXCLUSIVE = 1,
     PROTECTED = 3;
!
! OPEN^FILE FLAGS                  V         111111
!                                  0123456789012345
!                       !      1111111111222222233
DEFINE                  !4567890123456789012345678901
     ABORT^OPENERR = %B00000000000000000000000000001D#,
     ABORT^XFERERR = %B00000000000000000000000000010D#,
     PRINT^ERR^MSG = %B00000000000000000000000000100D#,
     AUTO^CREATE   = %B00000000000000000000000001000D#,
     MUSTBENEW     = %B00000000000000000000000010000D#,
     PURGE^DATA    = %B00000000000000000000000100000D#,
     AUTO^TOF      = %B00000000000000000000001000000D#,
     NOWAIT        = %B00000000000000000000010000000D#,
     BLOCKED       = %B00000000000000000000100000000D#,
     VAR^FORMAT    = %B00000000000000000001000000000D#,
     READ^TRIM     = %B00000000000000000010000000000D#,
     WRITE^TRIM    = %B00000000000000000100000000000D#,
     WRITE^FOLD    = %B00000000000000001000000000000D#,
     WRITE^PAD     = %B00000000000000010000000000000D#,
     CRLF^BREAK    = %B00000000000000100000000000000D#;
!
! sio procedure errors.
!
LITERAL
     SIOERR^INVALIDPARAM      = 512, ! parameter is invalid.
     SIOERR^MISSINGFILENAME   = 513, ! filename not supplied.
     SIOERR^DEVNOTSUPPORTED   = 514, ! device not supported.
     SIOERR^INVALIDACCESS     = 515, ! access mode incompatible with
                                     ! device.
     SIOERR^INVALIDBUFADDR    = 516, ! buffer address not in lower 32k.
     SIOERR^INVALIDFILECODE   = 517, ! file code of file does not match
                                     ! assigned file code.
     SIOERR^BUFTOOSMALL       = 518, ! buffer to small for edit write
                                     ! (ie., less than 1024 bytes) or
```

```
                                    ! buffer not sufficient for record
                                    ! length.
    SIOERR^INVALIDBLKLENGTH = 519,  ! assign block length > block
                                    ! buffer length.
    SIOERR^INVALIDRECLENGTH = 520,  ! record length = 0 or record
                                    ! length > maxrecordlength of
                                    ! OPEN^FILE or record length for
                                    ! $RECEIVE file < 14 ! or record
                                    ! length > 254 and variable
                                    ! records specified.
    SIOERR^INVALIDEDITFILE  = 521,  ! edit file is invalid.
    SIOERR^FILEALREADYOPEN  = 522,  ! OPEN^FILE called for file
                                    ! already open.
    SIOERR^EDITREADERR      = 523,  ! edit read error.
    SIOERR^FILENOTOPEN      = 524,  ! file not open.
    SIOERR^ACCESSVIOLATION  = 525,  ! access not in effect for
                                    ! requested operation.
    SIOERR^NOSTACKSPACE     = 526,  ! insufficient stack space for
                                    ! temporary buffer allocation.
    SIOERR^BLOCKINGREQD     = 527,  ! block buffer required for nowait
                                    ! fold or pad.
    SIOERR^EDITDIROVERFLOW  = 528,  ! edit write directory overflow.
    SIOERR^INVALIDEDITWRITE = 529,  ! write attempted after directory
                                    ! has been written.
    SIOERR^INVALIDRECVWRITE = 530,  ! write to $RECEIVE does not
                                    ! follow read.
    SIOERR^CANTOPENRECV     = 531,  ! can't open $RECEIVE for break
                                    ! monitoring.
    SIOERR^IORESTARTED      = 532,  ! nowait io restarted.
    SIOERR^INTERNAL         = 533;  ! internal error.
```

The following is the internal structure of the File Control Block (FCB).

Note:

> The FCB is included as a debugging aid only. TANDEM Computers, Incorporated, reserves the right to make changes to the FCB structure. Therefore, this information must not be used to make program references to elements within the File Control Block.

```
!
! File Control Block (FCB) Structure Template.
!
STRUCT FCB^TMPL ( * );
  BEGIN
    INT SIZE,                       ! ( 0) size of FCB in words.
        NAMEOFFSET,                 ! ( 1) word offset to name.
        FNUM;                       ! ( 2) guardian file number, -1 =
    !                                      closed.
    !
    ! create/open options group.
    !
    INT OPTIONS1,                   ! ( 3) assign options.
        OPTIONS2,                   ! ( 4) assign options.
        FILENAME [ 0:11 ],          ! ( 5) Tandem file name.
        !
        ! create options.
        !
        FCODE,                      ! (17) file code.
        PRIEXT,                     ! (18) primary extent size in pages.
        SECEXT,                     ! (19) secondary extent size in
        !                                  pages.
        RECLEN,                     ! (20) logical record length.
        BLKBUFLEN,                  ! (21) block length from ASSIGN,
        !                                  block buffer
        !                                  length following OPEN^FILE.
        !
        ! open options.
        !
        OPENEXCLUSION,              ! (22) exclusion bits to OPEN.
        OPENACCESS;                 ! (23) access bits to OPEN.
    !
    ! initializer group.
    !
    INT PUCB^POINTER,               ! (24) not used by sio procedures.
        SAMEFILELINK;               ! (25) not used by sio procedures.
    !
    ! beginning of sio groups.
    !
```

```
    INT FWDLINK,                ! (26) forward link.
        BWDLINK,                ! (27) backward link.
        ADDR,                   ! (28) address of this fcb.
        COMMONFCBADDR,          ! (29) address of common FCB.
        ERROR;                  ! (30) last error.
    !
    ! file FCB section.
    !
    INT DEVINFO,                ! (31) file type, dev type, dev
                                !      subtype.
        OPENFLAGS1,             ! (32) access mode, flags parameters
                                !      to OPEN^F&ILE.
        OPENFLAGS2,             ! (33) flags parameters to OPEN^FILE.
        XFERCNTL1,              ! (34) iotype, sysbuflen, interactive
                                !      prompt.
        XFERCNTL2,              ! (35) physioout, logioout, write
                                !      flush, retry count, edit write
                                !      control.
        DUPFCBADDR;             ! (36) FCB address of file where data
                                !      read from this file is to be
                                !      written.

    INT(32)
        LINENO;                 ! (37) line number from edit read or
                                !      ordinal record count scaled by
                                !      1000.
        !
        ! Data Transfer/Blocking Group.
        !
    INT BLKBUFADDR,             ! (39) word address of block buffer.
        BLKXFERCNT,             ! (40) number of bytes to be
                                !      transferred between device and
                                !      target buffer.
        BLKREADCNT =            ! (40) number of bytes to be read
          BLKXFERCNT,           !      from device to target buffer.
        BLKWRITECNT =           ! (40) number of bytes to be written
          BLKXFERCNT,           !      from target buffer to device.
        BLKCNTXFERRED,          ! (41) number of bytes transferred
                                !      between device and target
                                !      buffer.
        BLKCNTREAD =            ! (41) number of bytes read into
                                !      target buffer.
          BLKCNTXFERRED,
        BLKCNTWRITTEN =         ! (41) number of bytes written from
          BLKCNTXFERRED,        !      target buffer.
        BLKNEXTREC,             ! (42) (byte address) While blocking/
                                !      deblocking this is the address
                                !      of the next record pointer in
                                !      the block buffer.
        USRBUFADDR,             ! (43) byte address of user buffer.
        USRWRCNT,               ! (44) <write count> parameter of
                                !      WRITE^FILE, <prompt count>
                                !      parameter of READ^FILE.
```

```
    USRRDCNT,                   ! (45) <max read count> parameter of
                                !       READ^FILE.
    TFOLDLEN,                   ! (46) terminal write fold length
                                !      (= physical record length).
    USRCNTRD =                  ! (46) number of bytes read into user
                                !      buffer.
      TFOLDLEN,
    PHYSXFERCNT,                ! (47) transfer count value passed to
                                !      file system in SIO^PIO.
    PHYSIOCNTXFERRED,           ! (48) count transferred value
                                !      returned from file system
                                !      procedure.
    PHYSIOCNTRD =               ! (48) count read value returned from
       PHYSIOCNTXFERRED,        !      file system.
    PHYSIOCNTWR =               ! (48) count written value returned
       PHYSIOCNTXFERRED;        !      from file system.
!
INT USERFLAG =                  ! (24) flag word to be set by user.
    PUCB^POINTER;
!
! initializer group.
!
INT LOGICALFILENAME [ 0:3 ]; ! (49) logical file name of this file
                                !      to initializer.
!
! common FCB section.
!
    !
    ! Break Group.
    !
INT BRKFCBADDR =                ! (31) FCB of file owning break.
      DEVINFO,
    BRKMSG =                    ! (32:33) break message buffer.
      OPENFLAGS2,
    BRKCNTL =                   ! (34) break control.
      XFERCNTL1,
    BRKLASTOWNER =              ! (35) break last owner.
      XFERCNTL2;
    !
    ! $RECEIVE Group.
    !
INT SYSMSGS =                   ! (36) System messages to be
      DUPFCBADDR,               !      passed back to caller.
    RCVCNTL =                   ! (37) $RECEIVE control.
      LINENO,
    PRIMARYPID [-1:-1] =        ! (38:41) Primary opener's <process
                                !         id>.
      LINENO,
    BACKUPPID =                 ! (42:45) Backup opener's <process
                                !         id>.
      BLKNEXTREC,
    REPLYCODE =                 ! (46) $RECEIVE reply error code.
      TFOLDLEN;
```

```
        !
        ! Misc Group.
        !
    INT COMMCNTL =                  ! (47)
          PHYSXFERCNT,
        OPRQSTFCBADDR =             ! (48) FCB of file for which operator
          PHYSIOCNTXFERRED,         !      console messages are being
                                    !      displayed. (see NO^ERROR,
                                    !      prompt).
        OPRQSTCOUNT =               ! (49) count of number of operator
          LOGICALFILENAME,          !      messages displayed.  (see
                                    !      NO^ERROR, prompt).
        ERRFCBADDR [-1:-1] =        ! (50) FCB address of file where
          LOGICALFILENAME;          !      errors are to be reported.
    !
    INT SPARE1,                     ! (53) unused fcb word.
        SPARE2,                     ! (54) unused fcb word.
        SPARE3,                     ! (55) unused fcb word.
        SPARE4,                     ! (56) unused fcb word.
        SPARE5,                     ! (57) unused fcb word.
        SPARE6,                     ! (58) unused fcb word.
        SPARE7;                     ! (59) unused fcb word.
    !
  END; ! FCB^TMPL.
!
! -- BIT FIELDS.
!
! - ASSIGN BITS.
!
DEFINE
    !
    FILENAMESUPPLD  = <0>#,
      FCB^FILENAMESUPPLD      = FCB.OPTIONS1.FILENAMESUPPLD#,
    !
    PRIEXTSUPPLD    = <1>#,
      FCB^PRIEXTSUPPLD        = FCB.OPTIONS1.PRIEXTSUPPLD#,
    !
    SECEXTSUPPLD    = <2>#,
      FCB^SECEXTSUPPLD        = FCB.OPTIONS1.SECEXTSUPPLD#,
    !
    FCODESUPPLD     = <3>#,
      FCB^FCODESUPPLD         = FCB.OPTIONS1.FCODESUPPLD#,
    !
    EXCLUSIONSUPPLD = <4>#,
      FCB^EXCLUSIONSUPPLD     = FCB.OPTIONS1.EXCLUSIONSUPPLD#,
    !
    ACCESSSUPPLD    = <5>#,
      FCB^ACCESSSUPPLD        = FCB.OPTIONS1.ACCESSSUPPLD#,
    !
    RRECLENSUPPLD   = <6>#,
      FCB^RRECLENSUPPLD       = FCB.OPTIONS1.RRECLENSUPPLD#,
    !
    BLOCKLENSUPPLD  = <7>#,
      FCB^BLOCKLENSUPPLD      = FCB.OPTIONS1.BLOCKLENSUPPLD#;
```

```
!
! - OPEN EXCLUSION (FCB^OPENEXCLUSION)
!
DEFINE
     EXCLUSIONFIELD = <9:11>#,
       FCB^EXCLUSIONFIELD      = FCB.OPENEXCLUSION.EXCLUSIONFIELD#;
!
! - OPEN ACCESS (FCB^OPENACCESS)
!
DEFINE
     ACCESSFIELD    = <3:5>#,
       FCB^ACCESSFIELD         = FCB.OPENACCESS.ACCESSFIELD#;
!
! - DEVINFO.
!
DEFINE
     FILETYPE    =    <0:3>#,
       FCB^FILETYPE            = FCB.DEVINFO.FILETYPE#;
LITERAL
     UNSTR       = 0,
     ESEQ        = 1,
     REL         = 2,
     KSEQ        = 3,
     EDIT        = 4,
     ODDUNSTR    = 8;
DEFINE
     STRUCTFILE  = <2:3>#,       ! <>0 means structured file.
       FCB^STRUCTFILE          = FCB^DEVINFO.STRUCTFILE#;

DEFINE
     DEVTYPE     =    <4:9>#,
       FCB^DEVTYPE             = FCB.DEVINFO.DEVTYPE#;
LITERAL
     PROCESS     = 0,
     OPERATOR    = 1,
     RECEIVE     = 2,
     DISC        = 3,
     MAGTAPE     = 4,
     PRINTER     = 5,
     TERMINAL    = 6,
     DATACOMM    = 7,
     CARDRDR     = 8;
DEFINE
     DEVSUBTYPE = <10:15>#,
       FCB^DEVSUBTYPE          = FCB.DEVINFO.DEVSUBTYPE#;
!
! OPEN FLAGS. ( FCB.OPENFLAGS1 )
!
DEFINE
     FILECREATED       =  <0>#,  ! new file created.
       FCB^FILECREATED         = FCB.OPENFLAGS1.FILECREATED#;
DEFINE
     ACCESS         =    <1:3>#,  ! access mode.
       FCB^ACCESS              = FCB.OPENFLAGS1.ACCESS#;
```

```
LITERAL
       READACCESS      = 1,
       WRITEACCESS     = 2,
       READWRITEACCESS = 3;
!
! OPEN FLAGS. ( FCB.OPENFLAGS2 )
!
DEFINE
     ABORTONOPENERROR = <15>#,  ! abend on fatal error during open.
       FCB^ABORTONOPENERROR   = FCB.OPENFLAGS2.ABORTONOPENERROR#;
DEFINE
     ABORTONXFERERROR = <14>#,  ! abend on fatal error during data
                                ! transfer.
       FCB^ABORTONXFERERROR   = FCB.OPENFLAGS2.ABORTONXFERERROR#;
DEFINE
     PRINTERRMSG      = <13>#,  ! print error message on fatal error.
       FCB^PRINTERRMSG        = FCB.OPENFLAGS2.PRINTERRMSG#;
DEFINE
     AUTOCREATE       = <12>#,  ! create a file if write access.
       ! 0 = don't.
       ! 1 = do.
       FCB^AUTOCREATE         = FCB.OPENFLAGS2.AUTOCREATE#;
DEFINE
     FILEMUSTBENEW    = <11>#,  ! if autocreate = 1, no such file may
                                ! currently exist.
       ! 0 = old file is allowed.
       ! 1 = file must be new.
       FCB^FILEMUSTBENEW      = FCB.OPENFLAGS2.FILEMUSTBENEW#;
DEFINE
     WRITEPURGEDATA   = <10>#,  ! purge existing data.
       ! 0 = APPEND.
       ! 1 = PURGEDATA.
       FCB^WRITEPURGEDATA     = FCB.OPENFLAGS2.WRITEPURGEDATA#;
DEFINE
     AUTOTOF          =  <9>#,  ! auto page eject on open for
                                ! printer/process.
       ! 0 = NO
       ! 1 = YES
       FCB^AUTOTOF            = FCB.OPENFLAGS2.AUTOTOF#;
DEFINE
     NOWAITIO         =  <8>#,  ! open with nowait depth of 1.
       ! 0 = WAIT.
       ! 1 = NOWAIT.
       FCB^NOWAITIO           = FCB.OPENFLAGS2.NOWAITIO#;
DEFINE
     BLOCKEDIO        =  <7>#,  ! blocked io.
       ! 0 = NOT BLOCKED
       ! 1 = BLOCKED
       FCB^BLOCKEDIO          = FCB.OPENFLAGS2.BLOCKEDIO#;
DEFINE
     VARFORMAT        =  <6>#,  ! variable length records.
       ! 0 = FIXED LENGTH
       ! 1 = VARIABLE LENGTH
       FCB^VARFORMAT          = FCB.OPENFLAGS2.VARFORMAT#;
```

```
DEFINE
     READTRIM          =  <5>#,  ! trim trailing blanks.
       ! 0 = NOTRIM
       ! 1 = TRIM
       FCB^READTRIM            = FCB.OPENFLAGS2.READTRIM#;
DEFINE
     WRITETRIM         =  <4>#,  ! trim trailing blanks.
       ! 0 = NOTRIM
       ! 1 = TRIM
       FCB^WRITETRIM           = FCB.OPENFLAGS2.WRITETRIM#;
DEFINE
     WRITEFOLD         =  <3>#,  ! fold write transfers greater than
       ! 0 = TRUNCATE              write record length bytes into
       ! 1 = FOLD                  multiple records.
       FCB^WRITEFOLD           = FCB.OPENFLAGS2.WRITEFOLD#;
DEFINE
     WRITEPAD          =  <2>#,  ! pad record with trailing blanks.
       FCB^WRITEPAD            = FCB.OPENFLAGS2.WRITEPAD#;
DEFINE
     CRLFBREAK         =  <1>#,  ! carriage return/line feed on break.
       ! 0 = NO CRLF ON BREAK.
       ! 1 = CRLF ON BREAK.
       FCB^CRLFBREAK           = FCB.OPENFLAGS2.CRLFBREAK#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL1 )
!
DEFINE
     READIOTYPE       = <1:3>#,
       ! 0 = READ
       ! 1 = READUPDATE/REPLY
       ! 2 = EDITREAD
       ! 3 = WRITEREAD
       ! 7 = INVALID
       FCB^READIOTYPE          = FCB.XFERCNTL1.READIOTYPE#;
LITERAL
       STANDARDTYPE    = 0,
       RECEIVETYPE     = 1,
       EDITTYPE        = 2,
       INTERACTIVETYPE = 3,
       INVALIDTYPE     = 7;
DEFINE
     WRITEIOTYPE      = <4:6>#,
       ! 0 = WRITE
       ! 1 = READUPDATE/REPLY
       ! 2 = EDITWRITE
       ! 7 = INVALID
       FCB^WRITEIOTYPE         = FCB.XFERCNTL1.WRITEIOTYPE#;
DEFINE
     SYSBUFLEN        = <7:8>#,  ! system buffer length / 1024.
       FCB^SYSBUFLEN           = FCB.XFERCNTL1.SYSBUFLEN#;
DEFINE
     PROMPT          = <9:15>#,  ! interactive prompt character.
       FCB^PROMPT              = FCB.XFERCNTL1.PROMPT#;
```

```
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
!
DEFINE
     PHYSIOOUT          = <0>#,  ! physical [read/write] io
                                 ! outstanding.
       FCB^PHYSIOOUT            = FCB.XFERCNTL2.PHYSIOOUT#,
     READIOOUT          = <1>#,  ! logical read io outstanding.
       FCB^READIOOUT            = FCB.XFERCNTL2.READIOOUT#,
     WRITEIOOUT         = <2>#,  ! logical write io outstanding.
       FCB^WRITEIOOUT           = FCB.XFERCNTL2.WRITEIOOUT#,
     LOGIOOUT         = <1:2>#,  ! logical io outstanding.
       FCB^LOGIOOUT             = FCB.XFERCNTL2.LOGIOOUT#,
     WRITEFLUSH        =  <3>#,  ! block buffer flush operation in
                                 ! progress.
       FCB^WRITEFLUSH           = FCB.XFERCNTL2.WRITEFLUSH#,
     RETRYCOUNT       = <4:5>#,  ! i/o retry counter.
       FCB^RETRYCOUNT           = FCB.XFERCNTL2.RETRYCOUNT#,
     NOPARTIALREC       = <6>#,  ! blocks contain only full records.
       FCB^NOPARTIALREC         = FCB.XFERCNTL2.NOPARTIALREC#;
!
! TRANSFER CONTROL ( FCB.XFERCNTL2 )
!
! -- EDIT READ/WRITE CONTROL
!
DEFINE
     EDDIRWIP         =     <7>#, ! directory write in progress.
       FCB^EDDIRWIP             = FCB.XFERCNTL2.EDDIRWIP#,
     EDHALFSECTCNT  =  <8:11>#, ! number of half sectors written in
                                 ! current data page after next
                                 ! physical write.
       FCB^EDHALFSECTCNT        = FCB.XFERCNTL2.EDHALFSECTCNT#,
     EDDATABUFLEN   = <12:15>#, ! edit data buf size ">>"
                                 ! EDDBUFSHIFT (8).
       FCB^EDDATABUFLEN         = FCB.XFERCNTL2.EDDATABUFLEN#,
     EDREPOSITION   =     <7>#, ! user is repositioning edit file
                                 ! (read op).
       FCB^EDREPOSITION         = FCB.XFERCNTL2.EDREPOSITION#;
!
! WRITE^FILE CONTROL OPERATION IN PROGRESS ( FCB.PHYSXFERCNT )
!
DEFINE
     CNTLINPROGRESS =     <0>#,
       FCB^CNTLINPROGRESS       = FCB.PHYSXFERCNT.CNTLINPROGRESS#,
     FORMSCNTLOP    =  <1:15>#,
       FCB^FORMSCNTLOP          = FCB.PHYSXFERCNT.FORMSCNTLOP#;
!
! BREAK CONTROL ( COMMFCB.BRKCNTL )
!
```

```
DEFINE
     BRKLASTMODE      = <0>#,     ! last break mode from SETMODE.
       COMMFCB^BRKLASTMODE     = COMMFCB.BRKCNTL.BRKLASTMODE#,
     BRKHIT           = <1>#,     ! break key has been typed but not
                                  ! tested.
       COMMFCB^BRKHIT          = COMMFCB.BRKCNTL.BRKHIT#,
     BRKFLUSH         = <2>#,     ! flush $RECEIVE break message.
       COMMFCB^BRKFLUSH        = COMMFCB.BRKCNTL.BRKFLUSH#,
     BRKSTOLEN        = <3>#,     ! break stolen away by another
                                  ! process.
       COMMFCB^BRKSTOLEN       = COMMFCB.BRKCNTL.BRKSTOLEN#,
     BRKLDN           = <8:15>#, ! logical device number of terminal.
       COMMFCB^BRKLDN          = COMMFCB.BRKCNTL.BRKLDN#,
     COMMFCB^BRKARMED =           ! break is armed.
       COMMFCB^BRKFCBADDR#;
!
! $RECEIVE CONTROL ( COMMFCB.RCVCNTL )
!
DEFINE
     RCVDATAOPEN      =   <0>#, ! receive has been opened for data
                                  ! transfer.
       COMMFCB^RCVDATAOPEN     = COMMFCB.RCVCNTL.RCVDATAOPEN#,
     RCVBRKOPEN       =   <1>#, ! receive has been opened for break
                                  ! message reception.
       COMMFCB^RCVBRKOPEN      = COMMFCB.RCVCNTL.RCVBRKOPEN#,
     RCVOPENCNT       = <2:3>#, ! count of open messages received.
       COMMFCB^RCVOPENCNT      = COMMFCB.RCVCNTL.RCVOPENCNT#,
     RCVSTATE         =   <4>#,
       ! 0 = NEED READUPDATE.
       ! 1 = NEED REPLY.
       COMMFCB^RCVSTATE        = COMMFCB.RCVCNTL.RCVSTATE#,
     RCVUSEROPENREPLY =   <5>#, ! user will reply to open messages.
       ! 0 = SIO REPLYS.
       ! 1 = USER REPLYS.
       COMMFCB^RCVUSEROPENREPLY = COMMFCB.RCVCNTL.RCVUSEROPENREPLY#,
     RCVPSUEDOEOF     =   <6>#, ! pseudo-eof. (n.a. if user wants
                                  ! close messages)
       ! 0 = EAT CLOSE MESSAGE.
       ! 1 = TURN LAST CLOSE MESSAGE INTO EOF.
       COMMFCB^RCVPSUEDOEOF    = COMMFCB.RCVCNTL.RCVPSUEDOEOF#,
     MONCPUMSG        = <2:3>#, ! user cpu up/down messages.
       COMMFCB^MONCPUMSG       = COMMFCB.SYSMSGS.MONCPUMSG#,
     OPENMSG          =  <10>#, ! user wants open messages.
       COMMFCB^OPENMSG         = COMMFCB.SYSMSGS.OPENMSG#,
     CLOSEMSG         =  <11>#, ! user wants close messages.
       COMMFCB^CLOSEMSG        = COMMFCB.SYSMSGS.CLOSEMSG#;
```

```
!
! COMMON CONTROL ( COMMFCB.COMMCNTL )
!
DEFINE
     CREATEINPROGRESS  = <0>#,  ! 1 during call to OPEN^FILE while
                                ! creating.
       COMMFCB^CREATEINPROGRESS = COMMFCB.COMMCNTL.CREATEINPROGRESS#,
     OPENINPROGRESS    = <1>#,  ! 1 during call to OPEN^FILE.
       COMMFCB^OPENINPROGRESS = COMMFCB.COMMCNTL.OPENINPROGRESS#,
     OPTYPE          = <0:1>#,  ! operation type.
       COMMFCB^OPTYPE         = COMMFCB.COMMCNTL.OPTYPE#,
     DEFAULTERRFILE    = <2>#,  ! defines default error reporting
                                ! file.
       ! 0 = home terinal.
       ! 1 = operator ($0).
       COMMFCB^DEFAULTERRFILE = COMMFCB.COMMCNTL.DEFAULTERRFILE#,
     TRACEBACK         = <3>#,  ! 1 = trace back to callers p when
                                ! printing an error message.
       COMMFCB^TRACEBACK      = COMMFCB.COMMCNTL.TRACEBACK#;
```

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
!	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

→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
(	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	Semi-colon
<	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
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

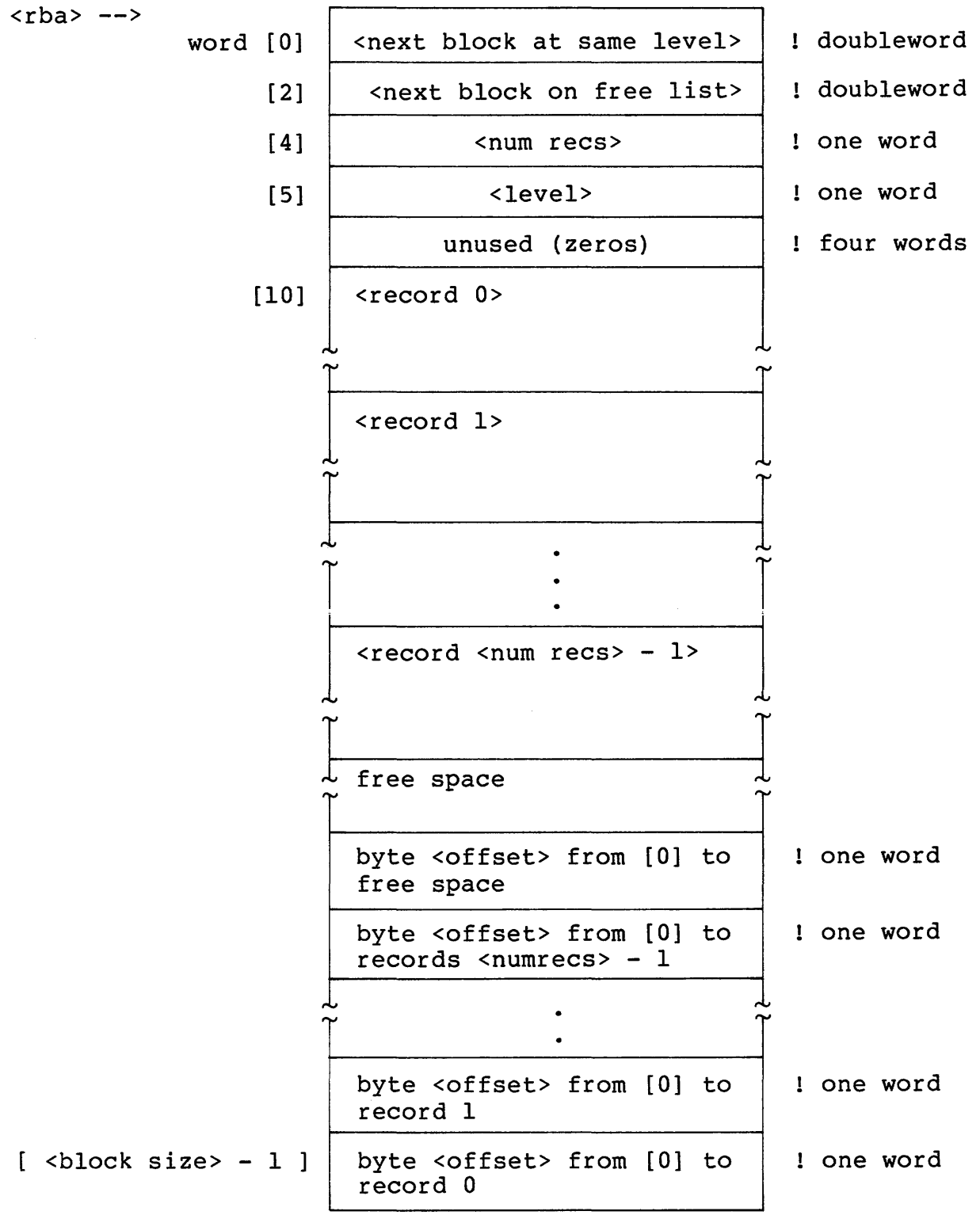
→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
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	Opening bracket
\	056000	000134	Left slant
]	056400	000135	Closing 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
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

→

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
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

ENSCRIBE STRUCTURED FILE BLOCK FORMAT



<rba>

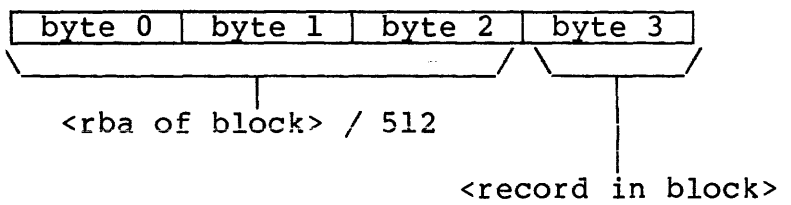
A block is addressed by a doubleword relative byte address. In a key-sequenced file, <rba> = 0D points to the root (highest level) index block; in a relative or entry-sequenced file, <rba> = 0D points to the first data block

To locate a given record in a key-sequenced file, the key value supplied to KEYPOSITION is used to search to the block for a record having a key field that matches

To locate a given record in a relative file, the <record number> supplied to [KEY]POSITION is converted to a block address and record number in the block as follows

```
<blocking factor> =
   ( <block length> - 22 ) / ( <create record length> + 2 )

<rba of block>    =
   <record number> / <blocking factor> * <block length>

<record in block> = <record number> '\' <blocking factor>
```

The format of a <record address> used to position to a record in an entry-sequenced file is

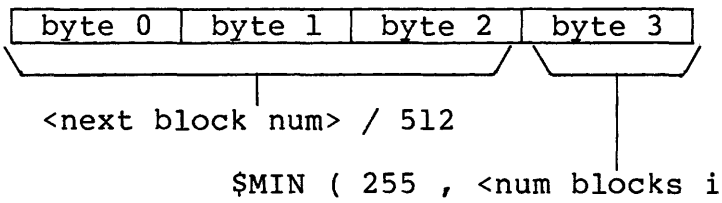


<next block at same level>,

for key-sequenced files, this is the block number of the next block at the same level; for relative and entry-sequenced files this field is not used and is set to zero

<next block on free list>,

for key-sequenced and relative files, this is the block number of the next block on the file's free list(s) (for a key-sequenced file, if the index and data block lengths differ, the file has two free lists; one for index blocks, one for data blocks) and the number of free blocks on the list. The doubleword is formatted as follows



For key-sequenced and relative files, if the entry = -1D then the block is the last block in the free list

For key-sequenced files, if the entry = -2D then the block is in use

For relative files, if the entry = -2D then there are no free records in the block

For entry-sequenced files, this entry is not used and is set to zero

<num recs>

is the number of records written in the block

<level>

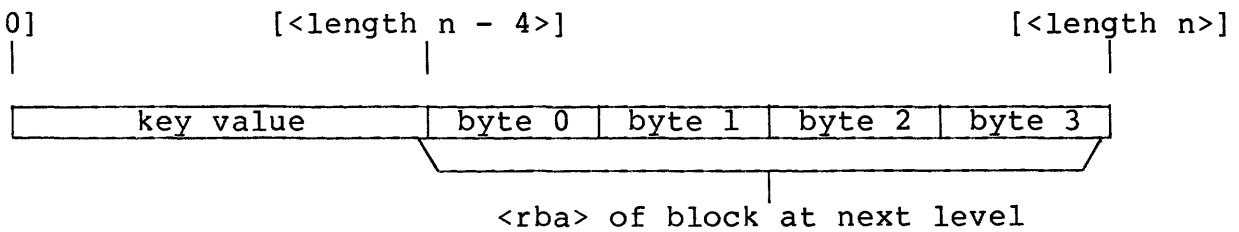
is the tree level of the block. <level> = 0 means that the block is a data block; <level> > 0 means that the block is an index block

<record>

is a data (if <level> = 0) or index (if <level> > 0) record. For a record "n",

<length n> = <offset to record n + 1> - <offset to record n>

The format of an index record is



<offset>

is the offest, in bytes, to the beginning of "free space" or a record. For relative files, the number of <offsets> is always the same as the <blocking factor>

<block size>

is the block size in words. This is calculated as follows

<block size> = (<create block length> + 1) / 2

The table on the following pages lists the File Management System errors individually by error number, giving a brief explanation of the meaning of each.

The <device type> that is associated with an error, as returned by the DEVICEINFO procedure is also given. The number corresponds to devices as follows:

<device type>

0	=	write to another process <process id>
1	=	operator console ($0)
2	=	$RECEIVE
3	=	Disc (3E = ENSCRIBE Structured File)
4	=	Magnetic Tape
5	=	Line Printer
6	=	Terminal: Conversational or Page Mode
7	=	Data Communications Line (ENVOY) 7.56 = Auto-Call Unit
8	=	Punched Card Reader
9	=	X.25 Access Method PTP Protocol
10	=	Data Communications Line (AXCESS)
11	=	Data Communications Line (ENVOY ACP)
12	=	Tandem to IBM Link (TIL)
20-23	=	Transaction Monitoring Facility (TMF)
26	=	Tandem Hyper Link

FILE MANAGEMENT ERROR LIST

<error>	description	<device type>
CCE		
0	operation successful	any
CCG		
1	end-of-file	3,4,6,8
2	operation not allowed on this type file	any
3	failure to open or purge a partition	3E
4	failure to open an alternate key file	3E
5	failure to provide sequential buffering	3E
6	system message received	2
7	process not accepting OPEN, CLOSE, CONTROL, or SETMODE messages	0
8	operation successful (examine MCW for additional status information)	7.*, 11.*
CCL		
10 (%12)	file/record already exists	3
11 (%13)	file not in directory or record not in file	3
12 (%14)	file in use	3 - 8
13 (%15)	illegal filename specification	any
14 (%16)	device does not exist	3 - 8
15 (%17)	volume specification supplied does not match name of volume on which the file actually resides	3
16 (%20)	file number has not been opened	any

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
17 (%21)	paired-open was specified and the file is not open by the primary process, the parameters supplied do not match the parameters supplied when the file was opened by the primary, or the primary process is not alive	any
18 (%22)	the referenced system does not exist	any
19 (%23)	no more devices in logical device table	3 - 8
20 (%24)	attempted network access by a process with a five-character name or a seven-character home terminal name	any
21 (%25)	illegal <count> specified	any
22 (%26)	application parameter or buffer address out of bounds	any
23 (%27)	illegal disc address	3
24 (%30)	privileged mode required for this operation	any
25 (%31)	AWAITIO or CANCEL attempted on "wait" file	any
26 (%32)	AWAITIO or CANCEL attempted on a file with no outstanding operations	any
27 (%33)	wait operation attempted when outstanding requests pending	any
28 (%34)	number of outstanding no-wait operations would exceed that specified at OPEN, or attempt to open a disc file or $RECEIVE with maximum number of concurrent operations greater than 1	any
29 (%35)	missing parameter	any
30 (%36)	unable to obtain main memory space for a link control block	0,1,3 - 8

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
31 (%37)	for Tandem NonStop System, unable to obtain SHORTPOOL space for a file system buffer area. For Tandem NonStop II System, unable to obtain file system buffer space.	any
32 (%40)	for Tandem NonStop System, unable to obtain main memory space for a control block; for Tandem NonStop II system, unable to obtain storage pool space (SYSPOOL); or, for either system, INFO proc. called with <file number> = -1 but no file was open.	any
33 (%41)	for Tandem NonStop System, i/o process is unable to obtain IOPOOL space for an i/o buffer, or, <count> too large for dedicated i/o buffer. For Tandem NonStop II System, i/o process is unable to obtain i/o segment space.	1, 3-12
34 (%42)	for Tandem NonStop II System only, unable to obtain file system control block	any
35 (%43)	for Tandem NonStop II System only, unable to obtain i/o process control block	1, 3-12
36 (%44)	for Tandem NonStop II System only, unable to lock physical memory	any
37 (%45)	for Tandem NonStop II System only, i/o process is unable to lock physical memory	1, 3-12
38 (%46)	for either Tandem NonStop System or Tandem NonStop II System, operation attempted on wrong type of system	any

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
40 (%50)	operation timed out. AWAITIO did not complete within the time specified by its <time limit> parameter. If a 0D <time limit> (completion check) or -1 <file number> (any file) was specified, then the operation is considered incomplete. Otherwise, the operation is considered completed	any
41 (%51)	checksum error on file synchronization block	3
42 (%52)	attempt to read from unallocated extent	3
43 (%53)	unable to obtain disc space for extent	3
44 (%54)	directory is full	3
45 (%55)	file is full	3
46 (%56)	invalid key specified	3E
47 (%57)	key not consistent with file data	3E
48 (%60)	security violation	3
49 (%61)	access violation	any
50 (%62)	directory error	3
51 (%63)	directory is bad	3
52 (%64)	error in disc free space table	3
53 (%65)	file system internal error	3
54 (%66)	i/o error in disc free space table	3
55 (%67)	i/o error in directory	3
56 (%70)	i/o error on volume label	3
57 (%71)	i/o error in file label	3
58 (%72)	disc free space table is bad	3

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
59 (%73)	file is bad	3
60 (%74)	volume on which this file resides has been removed, or device has been downed since the file was opened	3 - 8
61 (%75)	no file opens are permitted	3
62 (%76)	volume has been mounted, but mount order has not been given; file open not permitted	3
63 (%77) 64 (%100)	volume has been mounted and mount is in progress; file open not permitted	3
65 (%101)	only special requests permitted	3
66 (%102)	device has been downed by operator	1,3 - 8
70 (%106)	continue file operation	0,3
71 (%107)	duplicate record	3E
72 (%110)	attempt to access unmounted partition	3
73 (%111)	file/record locked	3
74 (%112)	READUPDATE called for $RECEIVE and number of messages queued exceeds <receive depth>, or REPLY called with invalid <message tag>, or REPLY called and no message is outstanding	2
75 (%113)	for Tandem NonStop Systems with TMF, requesting process has no current process TRANSID	20-23
76 (%114)	for Tandem NonStop Systems with TMF, TRANSID is in the process of ending	20-23
77 (%115)	for Tandem NonStop Systems with TMF, a TMF system file has the wrong file code	20-23
78 (%116)	for Tandem NonStop Systems with TMF, TRANSID is invalid or obsolete	20-23

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
79 (%117)	for Tandem NonStop Systems with TMF, attempt made by TRANSID to update or delete a record that it has not previously locked	20-23
80 (%120)	for Tandem NonStop Systems with TMF, invalid operation attempted on audited file or non-audited disc volume	20-23
81 (%121)	for Tandem NonStop Systems with TMF, attempted operation invalid for TRANSID that has outstanding no-wait i/o on a disc or process file	20-23
82 (%122)	for Tandem NonStop System with TMF, TMF is not running	20-23
83 (%123)	for Tandem NonStop System with TMF, process has initiated more concurrent transactions than can be handled	20-23
84 (%124)	for Tandem NonStop System with TMF, TMF is not configured	20-23
87 (%127)	waiting on a READ request and did not get it	sub-device 10
88 (%130)	a CONTROL READ is pending; new READ invalid	sub-device 10
89 (%131)	for Tandem NonStop System, READ after CONTROL complete came in too late. For Tandem NonStop II System, remote device has no buffer available.	sub-device 10
90 (%132)	for Tandem NonStop System with TMF, TRANSID aborted because its parent process died	20-23
92 (%134)	for Tandem NonStop System with TMF, TRANSID aborted because path to remote node is down	20-23

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
93 (%135)	for Tandem NonStop System with TMF, TRANSID aborted because it spanned too many audit files	20-23
94 (%136)	for Tandem NonStop System with TMF, TRANSID aborted by operator command	20-23
97 (%141)	for Tandem NonStop System with TMF, TRANSID was aborted	20-23
98 (%142)	for Tandem NonStop System with TMF, the Transaction Monitor Process's Network Active Transactions table is full	20-23
99 (%143)	attempt to use microcode option that is not installed	3
100 (%144)	device not ready	3,4,5,6,8
101 (%145)	no write ring	4
102 (%146)	paper out	5
103 (%147)	disc not ready due to power fail	3
110 (%156)	only break access permitted	6
111 (%157)	operation aborted because of break	6
112 (%160)	READ or WRITEREAD preempted by operator message	6
120 (%170)	data parity error	1,3 - 7
121 (%171)	data overrun error	1,3 - 8
122 (%172)	request aborted due to possible data loss caused by a reset of the circuit	6, 9
123 (%173)	sub-device busy	sub-device 10
124 (%174)	a line reset is in progress	sub-device 10 →

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
130 (%202)	illegal address to disc	3
131 (%203)	write check error from disc	3
132 (%204)	seek incomplete from disc	3
133 (%205)	access not ready on disc	3
134 (%206)	address compare error on disc	3
135 (%207)	write protect violation with disc	3
136 (%210)	unit ownership error (dual-port disc)	3
137 (%211)	controller buffer parity error	6.*, 7.*, 10.*, 11.*
140 8%214)	modem error (communication link not yet established, modem failure, momentary loss of carrier, or disconnect); for TIL, link disconnected	6,7
145 (%221)	card reader motion check error	8
146 (%222)	card reader read check error	8
147 (%223)	card reader invalid Hollerith code read	8
150 (%226)	end-of-tape marker detected	4
151 (%227)	runaway tape detected	4
152 (%230)	unusual end - tape unit went off line	4
153 (%231)	tape drive power on	4
154 (%232)	BOT detected during backspace files or backspace records	4
155 (%233)	only nine track tape permitted	4
156 (%234)	TIL protocol violation detected	12
157 (%235)	i/o process internal error	3 - 8

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
158 (%236)	invalid function requested for HyperLink	26
160 (%240)	request is invalid for line state	7.*, 11.*
	more than 7 reads or 7 writes issued	11.*
161 (%241)	impossible event occurred for line state	7.*, 11.*
162 (%242)	operation timed out	7.*, 11.*
163 (%243)	EOT received	7.0-7.3,7.8
	power at auto-call unit is off	7.56
164 (%244)	disconnect received	7.0-7.1, 11.*
	data line is occupied (busy)	7.56
165 (%245)	RVI received	7.0-7.3
	data line is not occupied after setting call request	7.56
166 (%246)	ENQ received	7.0-7.1, 7.3, 7.9
	auto-call unit failed to set "present next digit"	7.56
167 (%247)	EOT received on line bid/select	7.0-7.1, 7.3, 7.8
	"data set status" is not set after dialing all digits	7.56
168 (%250)	NAK received on line bid/select	7.0-7.1, 7.3, 7.8
	auto-call unit failed to clear "present next digit" after "digit present" was set	7.56

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
169 (%251)	WACK received on line bid/select	7.0-7.1,7.3
	auto-call unit set "abandon call and retry"	7.56
	station disabled or station not defined	11.*
170 (%252)	no ID sequence received during circuit assurance mode	7.0-7.1
	invalid MCW entry number on WRITE	11.40
171 (%253)	no response received or bid/poll/select	7.*, 10.*, 11.*
172 (%254)	reply not proper for protocol	7.*, 10.*, 11.*
173 (%255)	maximum allowable NAKs received	7.*, 10.*
	invalid MCW on WRITE	11.*
174 (%256)	WACK received after select	7.2-7.3
	aborted transmitted frame	11.*
175 (%257)	incorrect alternating ACK received	7.0-7.3
	command reject	11.*
176 (%260)	poll sequence ended with no responder	7.3,7.8-7.9
177 (%261)	text overrun	7.*, 10.*, 11.*
178 (%262)	no address list specified	7.2-7.3, 7.8-7.9, 11.40
179 (%263)	application buffer is incorrect	10.*
	control request pending or autopoll active	11.40
180 (%264)	unknown device status received	6.6-6.10, 10.* →

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
190 (%276)	invalid status received from device	1, 3-9, 10.*, 11.*
191 (%277)	device power on	5
192 (%300)	device is being exercised	3-6
200 (%310)	device is owned by alternate port	1, 3-9, 10.*, 11.*
201 (%311)	the current path to the device is down	1, 3-9, 10.*, 11.*
	an attempt was made to write to a non-existent process	0
210 (%322)	device ownership changed during operation	1, 3-9, 10.*, 11.*
211 (%323)	failure of cpu performing this operation	any
212 (%324)	For Tandem NonStop System only, EIO instruction failure	1, 3-9, 10.*, 11.*
213 (%325)	channel data parity error	1, 3-9, 10.*, 11.*
214 (%326)	channel timeout	1, 3-9, 10.*, 11.*
215 (%327)	i/o attempted to absent memory page	1, 3-9, 10.*, 11.*
216 (%330)	for Tandem NonStop System, map parity error during this i/o. For Tandem NonStop System, memory access breakpoint occurred during this i/o	1, 3-9, 10.*, 11.*
216 (%330)	map parity error during this i/o	1, 3-9, 10.*, 11.*
217 (%331)	memory parity error during this i/o	1, 3-9, 10.*, 11.*

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
218 (%332)	interrupt timeout	1, 3-9, 10.*, 11.*
219 (%333)	illegal device reconnect	1, 3-9, 10.*, 11.*
220 (%334)	protect violation	1, 3-9, 10.*, 11.*
221 (%335)	for Tandem NonStop System, pad-in violation. For Tandem NonStop II System, controller handshake violation.	1, 3-9, 10.*, 11.*
222 (%336)	bad channel status from EIO instruction	1, 3-9, 10.*, 11.*
223 (%337)	bad channel status from IIO instruction	1, 3-9, 10.*, 11.*
224 (%340)	controller error	1, 3-9, 10.*, 11.*
225 (%341)	no unit or multiple units assigned to same unit number	1, 3-9, 10.*, 11.*
230 (%346)	cpu power failed, then restored	1, 3-9, 10.*, 11.*
231 (%347)	controller power failed, then restored	1, 3-9, 10.*, 11.*
240 (%350)	network line handler error; operation not started	any
241 (%351)	network error; operation not started	any
248 (%370)	a line handler process failed while this request was outstanding. The file system recovers from this error for files opened with non-zero sync depth	any
249 (%371)	a network failure occurred while this request was outstanding. The file system recovers from this error for files opened with non-zero sync depth	any

→

FILE MANAGEMENT ERROR LIST (cont'd)

<error>	description	<device type>
250 (%372)	all paths to the system are down	any
251 (%373)	a network protocol error occurred	any
300-511	errors are reserved for process application-dependent usage	

APPENDIX E

ENSCRIBE CALL SYNTAX SUMMARY

```
CALL AWAITIO (     <file number>
               , [ <buffer address>    ]
               , [ <count transferred> ]
               , [ <tag>               ]
               , [ <time limit>        ]  )

CALL CANCEL ( <file number> )

CALL CANCELREQ   (    <file number>
                  , [ <tag>         ] )

CALL CLOSE ( <file number> )

CALL CONTROL (    <file number>
              ,   <operation>
              ,   <parameter>
              , [ <tag>         ] )

CALL CREATE (    <file name>
             , [ <primary extent size>   ]
             , [ <file code>             ]
             , [ <secondary extent size> ]
             , [ <file type>             ]
             , [ <record length>         ]
             , [ <data block length>     ]
             , [ <key-sequenced params>  ]
             , [ <alternate key params>  ]
             , [ <partition params>      ]  )

CALL DEVICEINFO (  <file name>
                 , <device type>
                 , <physical record length> )

<status> := EDITREAD (  <edit control block>
                      , <buffer>
                      , <buffer length>
                      , <sequence number>      )
```

```
<status> := EDITREADINIT (  <edit control block>
                          , <file number>
                          , <buffer length>       )


<status> := FILEERROR ( <file number> )


CALL FILEINFO (     <file number>
                , [ <error>                   ]
                , [ <file name>               ]
                , [ <logical device number>   ]
                , [ <device type>             ]
                , [ <extent size>             ]
                , [ <end-of-file location>    ]
                , [ <next-record pointer>     ]
                , [ <last mod time>           ]
                , [ <file code>               ]
                , [ <secondary extent size>   ]
                , [ <current-record pointer>  ]
                , [ <open flags>              ] )


CALL FILERECINFO (    <file number>
                  , [ <current key specifier>      ]
                  , [ <current key value>          ]
                  , [ <current key length>         ]
                  , [ <current primary key value>  ]
                  , [ <current primary key length> ]
                  , [ <partition in error>         ]
                  , [ <specifier of key in error>  ]
                  , [ <file type>                  ]
                  , [ <logical record length>      ]
                  , [ <block length>               ]
                  , [ <key-sequenced params>       ]
                  , [ <alternate key params>       ]
                  , [ <partition params>           ] )


{ length := }  FNAMECOLLAPSE (  <internal name>
                              , <external name> )


{ status := } FNAMECOMPARE (  <file name 1>
{ CALL     }                , <file name 2> )


{ <length> := } FNAMEEXPAND (   <external file name>
                              , <internal file name>
                              , <default names>      )
```

```
<status> := GETDEVNAME (    <logical device no>
                          ,   <device name>
                          , [ <system number>     ] )


CALL KEYPOSITION (    <file number>
                    ,   <key value>
                    , [ <key specifier>    ]
                    , [ <length word>      ]
                    , [ <positioning mode> ] )


CALL LOCKFILE (    <file number>
                 , [ <tag>         ] )


CALL LOCKREC (    <file number>
                , [ <tag>         ] )


<error> := NEXTFILENAME ( <file name> )


CALL OPEN (    <file name>
             ,   <file number>
             , [ <flags>                    ]
             , [ <sync depth>               ]
             , [ <primary file number>      ]
             , [ <primary process id>       ]
             , [ <sequential block buffer> ]
             , [ <buffer length>            ] )


CALL POSITION (  <file number>
                 , <record specifier> )


CALL PURGE ( <file name> )


CALL READ (    <file number>
             ,   <buffer>
             ,   <read count>
             , [ <count read>  ]
             , [ <tag>         ]  )
```

```
CALL READLOCK (    <file number>
               ,   <buffer>
               ,   <read count>
               , [ <count read>  ]
               , [ <tag>         ]  )

CALL READUPDATE (    <file number>
                 ,   <buffer>
                 ,   <read count>
                 , [ <count read>  ]
                 , [ <tag>         ] )

CALL READUPDATELOCK (    <file number>
                     ,   <buffer>
                     ,   <read count>
                     , [ <count read>  ]
                     , [ <tag>         ] )

CALL REFRESH [ ( $<volume name> ) ]

CALL RENAME (  <file number>
            , <new name> )

CALL REPOSITION (  <file number>
                , <positioning block> )

CALL SAVEPOSITION (    <file number>
                   ,   <positioning block>
                   , [ <positioning block size> ] )

CALL SETMODE (    <file number>
              ,   <function>
              , [ <parameter 1>  ]
              , [ <parameter 2>  ]
              , [ <last params>  ] )

CALL SETMODENOWAIT (    <file number>
                    ,   <function>
                    , [ <parameter 1> ]
                    , [ <parameter 2> ]
                    , [ <last params> ]
                    , [ <tag>         ]  )
```

```
CALL UNLOCKFILE (    <file number>
                  , [ <tag>         ] )

CALL UNLOCKREC (    <file number>
                 , [ <tag>         ] )

CALL WRITE (    <file number>
             ,  <buffer>
             ,  <write count>
             , [ <count written> ]
             , [ <tag>           ] )

CALL WRITEUPDATE (    <file number>
                   ,  <buffer>
                   ,  <write count>
                   , [ <count written> ]
                   , [ <tag>           ] )

CALL WRITEUPDATEUNLOCK (    <file number>
                         ,  <buffer>
                         ,  <write count>
                         , [ <count written> ]
                         , [ <tag>           ] )
```

INDEX

Access
  completion order, with AWAITIO 3-17
  description
    structured files 2-8
    unstructured files 2-18
  mode
    checking with OPEN 3-95
    description 1-14, 3-8
    setting with OPEN 3-89
  path
    description 2-9
    setting with KEYPOSITION 3-76
    setting with OPEN 3-97
    setting with POSITION 3-100
  positioning
    by alternate key for structured files 3-72
    by primary key for entry-sequenced 3-98
    by primary key for key-sequenced 3-72
    by primary key for relative 3-98
    by rba for unstructured files 3-98
    detailed description 4-43
  random
    for structured files 4-3
    for unstructured files 4-16
  relational, among structured files 1-11
  rules
    for structured files 4-2
    for unstructured files 4-7
  security
    checking with OPEN 3-94
    description 3-8
  sequential
    for structured files 4-2
    for unstructured files 4-12
  subsets, description 2-11

Alternate key
   attributes 2-14
   CREATE parameter array 5-34
   creation considerations
      automatic update suppression 5-10
      key length 5-9
      key offset 5-9
      key specifier 5-8
      null value 5-9
      uniqueness 5-10
   description 2-8, 2-12, 5-11
   file 2-13, 5-11
      creation with CREATE 5-28
      creation with FUP SET 5-14
      loading 6-6
      opening considerations 4-2
      record description 5-11
   for entry-sequenced files, description 2-15
   for key-sequenced files, description 2-14
   for relative files, description 2-14
   illustration of 2-13
   offset overlapping 5-9
   update, automatic 1-12
      suppression with CREATE 5-34
      supression with FUP SET 5-14
Approximate, access subset 2-11
   action during sequential reads 3-105
   positioning mode, overview 1-10
ASCII character set B-1
Automatic
   alternate key maintenance 1-12
      supression 5-10
   refresh, specification with CREATE 3-29
   refresh, specification with FUP SET 5-14
AWAITIO procedure
   completion chart 3-19
   error recovery 3-17
   FILEINFO response 3-18
   flowchart 3-20
   syntax description 3-14

Backup process, opening a file 3-91
Block, data see data block
Block, index: see index block
Buffer parameter 3-5
Buffer, cache: see cache buffer
Buffer, sequential block
   specifying with OPEN 3-91
BUILDKEYRECORDS, FUP command
   description 6-1
   example 6-10
   syntax description 6-9

Cache buffer 1-20
   usage tradeoff with index block length 5-8
CANCEL procedure
   completion 3-4
   syntax description 3-21
CANCELREQ procedure
   syntax description 3-22
CCE, condition code 3-6
CCG, condition code 3-6
CCL, condition code 3-6
Character set, ASCII B-1
CHECK^BREAK Procedure, SIO A-5
CHECK^FILE procedure, SIO A-6
CHECK^FILE Procedure, SIO
   example A-13
   file types A-9
CHECK^FILE procedure, SIO
   operations A-6
CLOSE procedure
   completion 3-4
   syntax description 3-23
CLOSE^FILE Procedure, SIO A-14
Code, file
   description 5-3
   specifying at creation
      with CREATE 5-28
      with FUP SET 5-14
Commands, FUP
   BUILDKEYRECORDS 6-9
   CREATE 5-24
   INFO 5-26
   LOAD 6-2
   LOADALTFILE 6-6
   RESET 5-25
   SET 5-14
   SHOW 5-23
Compare length
   setting prior to access with KEYPOSITION 3-73
Compare length for access subset 2-11
Compression, data and index
   description 5-6
   specification with CREATE 5-28
   specification with FUP SET 5-14
Condition codes
   returned from procedure call 3-6
Control block
   access (ACB), for partitioned files 3-96
   file (FCB)
      for partitioned files 3-96
      maintenance 3-24, 3-117

CONTROL procedure
   completion 3-4
   operations 3-25
   syntax description 3-24
CREATE procedure
   completion 3-4
   syntax description 5-28
Creating disc files 5-1
Current access path 2-9
   after successful OPEN 3-97
Current key specifier 2-9
   returned by FILERECINFO 3-56
Current key value
   positioning action 2-10
   returned by FILERECINFO 3-56
   setting, description 4-4
   specification with KEYPOSITION 2-10, 3-72
   specification with POSITION 2-10, 3-98
Current position 2-10
   after successful OPEN 3-97
   change after READ, example 2-10
   ENSCRIBE procedures that modify 4-4
   update by KEYPOSITION 3-76
Current state indicators 3-8
   status
      after KEYPOSITION 3-76
      after POSITION 3-100
      after READ 3-106
      after READUPDATE 3-114
      after WRITE 3-134
      after WRITEUPDATE 3-139

Data and index compression
   see: compression, data and index
Data block 2-2
   compression 2-3
      considerations 5-6
      specification with CREATE 5-28
      specification with FUP SET 5-14
   illustration
      for entry-sequenced file 2-7
      for key-sequenced file 2-4
      for relative file 2-6
   length, considerations 5-5
   size, specification
      with CREATE 5-28
      with FUP SET 5-14
   structured disc file format C-1
Data Definition Language (DDL) 1-22
Data, loading 6-1
   alternate key file 6-6
   primary file 6-2

DDL 1-22
Deadlock 4-26
Declarations, external
   ENSCRIBE procedures 3-9
Deletion
   file 3-101
DEVICEINFO procedure
   completion 3-4
   device types and subtypes 3-40
   syntax description 3-39
Disc
   device type 3-40
   files: see File, disc
   sectors 4-18
   volume, finding filenames on 3-86

EDITREAD procedure
   syntax description 3-42
EDITREADINIT procedure
   syntax description 3-46
End-of-file pointer
   for structured files 4-5
   for unstructured files 4-9
ENSCRIBE
   disc files: see File, disc
   file structures
      structured files 1-3
      unstructured files 1-13
   list of features 1-1
   procedure syntax summary E-1
   relationship to GUARDIAN 1-1
   structured file type comparison chart 2-15
   syntactic conventions i-9
ENSCRIBE procedures
   AWAITIO 3-14
   CANCEL 3-21
   CANCELREQ 3-22
   CLOSE 3-23
   CONTROL 3-24
   CREATE 3-27
   DEVICEINFO 3-39
   EDITREAD 3-42
   EDITREADINIT 3-46
   FILEERROR 3-48
   FILEINFO 3-51
   FILERECINFO 3-56
   FNAMECOLLAPSE 3-59
   FNAMECOMPARE 3-61
   FNAMEEXPAND 3-65
   GETDEVNAME 3-70

ENSCRIBE procedures (cont)
KEYPOSITION 3-72
LOCKFILE 3-78
LOCKREC 3-82
NEXTFILENAME 3-86
OPEN 3-88
POSITION 3-98
PURGE 3-101
READ 3-103
READLOCK 3-108
READUPDATE 3-111
READUPDATELOCK 3-115
REFRESH 3-117
RENAME 3-118
REPOSITION 3-120
SAVEPOSITION 3-121
SETMODE 3-123
SETMODENOWAIT 3-125
syntactic conventions i-9
syntax summary E-1
UNLOCKFILE 3-129
UNLOCKREC 3-130
WRITE 3-132
WRITEUPDATE 3-136
WRITEUPDATEUNLOCK 3-140
Entry-sequenced file 1-6, 2-7
alternate key, description 2-15
data block, illustration 2-7
positioning
by alternate key 3-72
by primary key 3-98
description 2-7
of record written to file 3-132
primary key assignment, description 2-7
record length, description 2-7
record length, maximum 5-5
specifying at creation
with CREATE 5-28
with FUP SET 5-14
structure, illustration 1-6
suggestions for use 2-7
Errors
file management
complete list D-1
general information 3-7
listed by number w/description 4-36
recovery considerations 4-39
Sequential I/O procedures A-40

Exact, access subset
   action during sequential reads 3-106
   positioning mode, overview 1-10
   subset of a file, description 2-11
Examples
   action of current position 4-48
   add new altkey to file w/out altkey file 6-14
   add new altkey to file with altkey file 6-13
   adding record to end of entry-sequenced 4-63
   adding record to end of relative file 4-62
   adding record within a relative file 4-62
   approx subset by alternate key 4-50
   approx subset by primary key 4-49
   creation of alternate key file 5-44
   creation of key-sequenced file 5-40
   creation of key-sequenced w/altkey 5-42
   creation of key-sequenced, partitioned 5-46
   creation of relative, partitioned 5-45
   exact subset by non-unique altkey 4-53
   exact subset by primary key 4-52
   generic subset by primary key 4-51
   load a key-sequenced file 6-12
   positioning of entry-sequenced file 4-60
   positioning of relative file 4-60
   random deletion with primary key 4-58
   random insertion for relative file 4-61
   random update to non-existent record 4-56
   random update, key-sequenced file 4-55
   record insertion, key-sequenced file 4-54
   relational processing 4-64
   reload partition of altkey file 6-16
   reload partition of key-seq file 6-15
   sequential read by pri-key w/delete 4-59
   sequential read by pri-key w/update 4-57
   sequential read of entry-sequenced file 4-61
   sequential read of relative file 4-61
Exclusion mode
   description 3-94
   setting with OPEN 3-89
Exclusion/Access Mode Checking 3-95
Extent
   allocation/deallocation 3-25
   considerations for partitioned files 5-4
   description 5-3
   size, primary, specifying at creation
      with CREATE 5-28
      with FUP SET 5-14
   size, secondary, specifying at creation
      with CREATE 5-28
      with FUP SET 5-14

External declarations
   ENSCRIBE procedures 3-9
   Sequential I/O A-4

Features of ENSCRIBE 1-1
File
   management errors 3-7
      complete list D-1
File Control Block Structure, SIO
   description A-43
   format A-70
   initializing the file FCB A-44
File management procedures
   see ENSCRIBE procedures 0
File System, theory of operation 1-24
File, disc 2-8
   access
      by alternate key for structured files 3-72
      by primary key for entry-sequenced 3-98
      by primary key for key-sequenced 3-72
      by primary key for relative 3-98
      by rba for unstructured files 3-98
      completion order for no-wait i/o 3-17
      mode and security checking 3-8
      mode: security checking 3-93
      mode: see Access mode
      path: see Access path
      relational, among structured files 1-11
      rules for structured files 4-2
      rules for unstructured files 4-7
      sharing 3-89
      subsets, description 1-10, 2-11
   alternate key: see alternate key
   code
      description 5-3
      specifying with CREATE 5-28
      specifying with FUP SET 5-14
   compression 2-3
      considerations 5-6
      with CREATE 5-28
      with FUP SET 5-14
   control block (FCB), maintenance 3-24, 3-117
   creation 5-1
   creation parameters
      returned by FILERECINFO 3-56
      setting with CREATE 5-28
      setting with FUP SET 5-14
   deletion 3-101
   determining status of 3-39
   entry-sequenced 1-6, 2-7
      see also entry sequenced file

File, disc (cont)
exclusion/access mode table 3-95
extent allocation/deallocation 3-25
key-sequenced 1-4, 2-2
see also key-sequenced file
loading data
alternate key file 6-6
general info 6-1
primary file 6-2
locking
detailed description 4-23
unlocking with UNLOCKFILE 3-129
with LOCKFILE 3-78
management procedures
see ENSCRIBE procedures
name: see Filename
number
assignment by operating system 3-92
parameter 3-5
returned from OPEN 3-92
opening, considerations 4-2
ownership
changing for an existing file 3-127
current status returned by SETMODE 3-124
partitioning across volumes/network node 1-21
array format 5-37
open considerations 4-2
specification with CREATE 5-28
specification with FUP SET 5-14
thorough description 5-4
purge security check 3-8
purging data 3-25
read security check 3-8
relative 1-5, 2-5
see also relative file
structure, illustration 1-5
renaming 3-118
security
changing security of existing file 3-127
checking, access table 3-94
current status returned by SETMODE 3-124
description 3-93
shared access 3-89
structured: block format C-1

File, disc (cont)
   subset 1-10, 2-11
      sequential reading of 3-103
   temporary
      name format 3-10
   type, specifying at creation
      with CREATE 5-28
      with FUP SET 5-14
   unlocking 3-129
   write security check 3-8
   writing a record 3-132
File/record locking interaction 4-24
FILEERROR procedure
   syntax description 3-48
FILEINFO procedure
   completion 3-4
   example
      random update to non-existent record 4-56
   syntax description 3-51
Filename
   comparing two for equivalence 3-61
   converting from external to internal 3-65
   converting from internal to external 3-59
   finding device type & physical rec len 3-39
   finding next in subvolume 3-86
   network form 3-12
   permanent form 3-10
   specifying with CREATE 5-28
   specifying with FUP SET 5-14
   temporary form 3-10
FILERECINFO procedure
   completion 3-4
   syntax description 3-56
Finding records 1-23
Flags, OPEN
   returned by FILEINFO 3-54
   setting with OPEN 3-89
FNAMECOLLAPSE procedure
   syntax description 3-59
FNAMECOMPARE procedure
   syntax description 3-61
FNAMEEXPAND procedure
   syntax description 3-65

FUP commands
   BUILDKEYRECORDS 6-9
   CREATE 5-24
   functions described 5-12
   INFO 5-26
   LOAD 6-2
   LOADALTFILE 6-6
   RESET 5-25
   SET 5-14
   SHOW 5-23
FUP, running 5-13

Generic, access subset
   action during sequential reads 3-106
   positioning mode, overview 1-10
   subset of a file, description 2-11
GETDEVNAME procedure
   syntax description 3-70
GIVE^BREAK Procedure, SIO A-16
Global declarations
   ENSCRIBE procedures 3-9
GUARDIAN
   file management error list D-1
   relationship to ENSCRIBE 1-1

I/O device control operations 3-25
Index block 2-2
   compression 2-3
      description 5-6
      specification with CREATE 5-28
      specification with FUP SET 5-14
   creation considerations 5-8
   diagram 2-4
   length, considerations 5-8
   size, specification at creation
      with CREATE 5-28
      with FUP SET 5-14
Inserting records 1-23
   considerations for structured files 4-3
   with WRITE 3-132
Interface with INITIALIZER and
   ASSIGN message, SIO A-48
      considerations A-51
      INITIALIZER-related defines A-48
      setting file characteristics A-51
      usage example A-52

Key field
   description 2-8
   illustration 2-9
Key length
   current, returned by FILERECINFO 3-56
   description 5-9
   specification at creation
      with CREATE 5-28
      with FUP SET 5-14
Key offset 5-9
   specification at creation
      with CREATE 5-28
      with FUP SET 5-14
Key specifier 2-9
   description 5-8
   example 2-9
Key value, current 2-10
Key, alternate: see alternate key
Key, term defined 2-8
Key-sequenced file 1-4, 2-2
   alternate key, description 2-14
   creation
      parameter array 5-33
      with CREATE 5-28
      with FUP 5-14
   data block, illustration 2-4
   data block: see data block
   index block: see index block
   number of reads per access 2-2
   positioning 3-72
   primary key, description 2-2
   record
      length, description 2-2
      length, maximum 5-5
   storage organization
      data block 2-2
      index block 2-2
KEYPOSITION procedure
   completion 3-4
   example
      action of current postition 4-48
      approx subset by alternate key 4-50
      exact subset by non-unique altkey 4-53
      exact subset by primary key 4-52
      generic subset by primary key 4-51
      random deletion with primary key 4-58
      random update to non-existent record 4-56
      random update, key-sequenced file 4-55
      sequential read by pri-key w/delete 4-59
      sequential read by pri-key w/update 4-57

KEYPOSITION procedure (cont)
   Repositioning to next record 3-77
   Repositioning to same record 3-77
   syntax description 3-72

Length, key: see key length
Length, record: see record length
LOAD, FUP command
   description 6-1
   example 6-3
   syntax description 6-2
LOADALTFILE, FUP command
   description 6-1
   example 6-6
   syntax description 6-6
Loading files
   alternate key files 6-6
   overview 1-22
   primary files 6-2
LOCKFILE procedure
   completion 3-4
   example 3-81
   setting lockmode 3-128
   syntax description 3-78
Locking
   deadlock situation 4-26
   detailed description 4-23
   file 3-78
   file/record locking interaction 4-24
   modes, alternate and default 4-24
   record 3-82
LOCKREC procedure
   completion 3-4
   dependency on current position 2-10
   setting lockmode 3-128
   syntax description 3-82
Logical records 5-5
LRU for cache buffer 1-20

Mirror volumes 1-44
Multi-key access to records 1-7
Multiple volume files 1-21
   open considerations 4-2
   specification at creation
      with CREATE 5-28
      with FUP SET 5-14
   thorough description 5-4

Network file names 3-12
NEXTFILENAME procedure
  completion 3-4
  example 3-87
  syntax description 3-86
No-wait i/o
  access completion 3-14
  description 1-16
Notational conventions i-9
NOWAIT I/O with SIO A-65
NO^ERROR Procedure, SIO A-58
  error handling A-60
Null value 2-14
  considerations with BUILDKEYRECORDS 6-11
  description 5-9

ODDUNSTR
  specifying with CREATE 5-28
  specifying with FUP SET 5-14
Offset, key: see key offset
Open flags
  returned from FILEINFO 3-54
  setting with OPEN 3-88
OPEN procedure
  by backup process 3-91
  completion 3-4
  condition code setting 3-92
  example
    approx subset by primary key 4-49
  flags
    returned by FILEINFO 3-54
    setting 3-89
  syntax description 3-88
OPEN^FILE Procedure, SIO A-17
  example A-22
  flags A-20
Ownership, file
  changing an existing file 3-127
  current status returned by SETMODE 3-124

Parameters, ENSCRIBE procedure
  buffer 3-5
  file number 3-5
  tag 3-5
  transfer count 3-6
Partitioned files 1-21
  open considerations 4-2
  opening of 3-96
  specification at creation
    with CREATE 5-28
    with FUP SET 5-14
  thorough description 5-4

Permanent file names 3-11
Physical record, description 1-3
POSITION procedure
  completion 3-4
  example
    for relative and entry-sequenced file 4-60
  syntax description 3-98
Positioning 2-11
  by alternate key for structured files 3-72
  by primary key for entry-sequenced files 3-98
  by primary key for key-sequenced files 3-72
  by primary key for relative files 3-98
  by rba key for unstructured files 3-98
  implementation details 4-43
  status
    after KEYPOSITION 3-76
    after POSITION 3-100
    after READ 3-106
    after READUPDATE 3-114
    after WRITE 3-134
Positioning mode 1-10
  approximate 2-11
    action during read 3-105
  exact 2-11
    action during read 3-106
  generic 2-11
    action during read 3-106
  setting prior to access with KEYPOSITION 3-74
  setting prior to access with POSITION 3-100
Primary extent size
  specifying with CREATE 5-28
  specifying with FUP SET 5-14
Primary file number, specifying w/ OPEN 3-88
Primary key 1-7, 1-9, 2-8
  creation considerations 5-7
  current value update
    for relative file 3-98
  specification at creation
    with CREATE 5-28
    with FUP SET 5-14
Primary process id, specifying w/ OPEN 3-91
Procedure call errors: see errors
Procedure parameters
  buffer 3-5
  file number 3-5
  tag 3-5
  transfer count 3-6

Procedures, ENSCRIBE
   see ENSCRIBE procedures
Process
   device type 3-40
   suspension for file access 3-14
Purge
   data within a file 3-25
   file from system 3-101
PURGE procedure
   completion 3-4
   syntax description 3-101
Purge, security check 3-8

Random access
   for structured files 4-3
   for unstructured files 4-16
READ procedure
   action following successful OPEN 3-97
   completion 3-4
   dependency on current position 2-10
   example
      approx subset by primary key 4-49
      approximate subset by alternate key 4-50
      exact subset by non-unique altkey 4-53
      exact subset by primary key 4-52
      generic subset by primary key 4-51
      sequential read by pri-key w/delete 4-59
      sequential read by pri-key w/update 4-57
   syntax description 3-103
Read, security check 3-8
Reading records, general info 1-23
READLOCK procedure
   completion 3-4
   dependency on current position 2-10
   example 3-109
   setting lockmode 3-128
   syntax description 3-108
READUPDATE procedure
   completion 3-4
   dependency on current position 2-10
   example
      random update to non-existent record 4-56
      random update, key-sequenced file 4-55
   syntax description 3-111
READUPDATELOCK procedure
   completion 3-4
   dependency on current position 2-10
   setting lock mode 3-128
   syntax description 3-115

READ^FILE Procedure, SIO A-23
Record 2-8
deblocking 1-20
deletion 1-23
identification within a file 1-7
insertion 1-23
considerations 4-3
with WRITE 3-132
length
specification with CREATE 5-28
specification with FUP SET 5-14
length, maximum
for entry-sequenced files 5-5
for key-sequenced files 5-5
for relative files 5-5
for unstructured files 3-29
locking 1-23
detailed description 4-23
during sequential reads (READLOCK) 3-108
for unstructured files 4-26
limitations 4-27
with LOCKREC 3-82
with READLOCK 3-108
with READUPDATELOCK 3-115
logical, description 5-5
management functions, overview 1-23
multi-key access to 1-7
physical length determination 3-39
physical, description 1-3
reading, general info 1-23
specifier 3-98
unlocking 3-130
general info 1-23
with UNLOCKFILE 3-129
with UNLOCKREC 3-130
with WRITEUPDATEUNLOCK 3-140
updating, general info 1-23
Recovery from errors 4-39
REFRESH procedure
syntax description 3-117
refresh, automatic
specification with CREATE 3-29
specification with FUP SET 5-14
Relational access among stuctured files 1-11
Relative byte addresses (rba)
description 2-18, 4-9
setting with POSITION 3-98

Relative file 1-5, 2-5
   data block diagram 2-6
   data block, illustration 2-6
   positioning
      by alternate key 3-72
      by primary key 3-98
      of record written to file 3-132
   primary key assignment, description 2-5
   record length, maximum 5-5
   specifying at creation
      with CREATE 5-28
      with FUP SET 5-14
   suggestions for use 2-6
RENAME procedure
   completion 3-4
   syntax description 3-118
Renaming files: see RENAME
REPOSITION procedure
   syntax description 3-120
Resident buffering (TNS only) 4-20

SAVEPOSITION procedure
   syntax description 3-121
Secondary extent size, specifying
   with CREATE 5-28
   with FUP SET 5-14
Security, file
   changing of an existing file 3-127
   checking 3-8
   checking, access table 3-94
   current status returned by SETMODE 3-124
Sequential access
   for structured files 4-2
   for unstructured files 4-12
Sequential access buffering option 1-20
   example 4-6
   specifying with OPEN 3-91
   thorough description 4-6
Sequential I/O Procedures
   call summary A-3
   errors A-40
   FCB Structure
      description A-43
      format A-70
      Initializing the File FCB A-44
   Interface with INITIALIZER and
      ASSIGN messages A-48

Sequential I/O Procedures (cont)
   Interface with INITIALIZER and ASSIGN
      considerations A-51
      INITIALIZER-related defines A-48
      summary A-55
      usage example A-48
   introduction A-1
SETMODE procedure
   completion 3-4
   syntax description 3-123
SETMODE/SETMODENOWAIT
   functions 3-127
SETMODENOWAIT procedure
   completion 3-4
   syntax description 3-125
SET^FILE Procedure, SIO A-25
   BREAK operation A-34
   NOWAIT I/O operations A-33
   operations A-26
Shared access, for a file 1-14, 3-89
Specifier, record 3-98
State indicators: see Current state ind.
Structured files 1-3
   comparison chart for 2-15
   see also Entry-sequenced file 0
   see also Key-sequenced file 0
   see also relative file 0
Subset, file 1-10
   description 2-12
   sequential reading of 3-103
Sync depth
   considerations for files w/alternate key 4-42
   specifying with OPEN 3-90
Syntactic conventions i-9

Tag parameter, general info 3-5
TAKE^BREAK Procedure, SIO A-35
Temporary file
   name format 3-10
   results of close 3-23
Temporary file names 3-11
Theory of operation, file system 1-24
Transfer count parameter 3-6

UNLOCKFILE procedure
   completion 3-4
   syntax description 3-129
Unlocking
   file 3-129
   record
      with UNLOCKREC 3-130
      with WRITEUPDATEUNLOCK 3-140
UNLOCKREC procedure
   completion 3-4
   syntax description 3-130
Unstructured files
   appending to end-of-file 4-16
   characteristics 2-16
   locking
      with LOCKFILE 3-78
   locking records
      description 3-85
      with LOCKREC 3-130
      with READLOCK 3-108
      with READUPDATELOCK 3-115
   overview 1-13
   positioning
      description 2-18
      with POSITION 3-98
   random access to 4-16
   relative byte addresses 2-18, 4-9
   resident buffering 4-20
   sequential access to 4-12
Updating records 1-23
   procedure to accomplish 3-132
Usage Example, SIO
   with INITIALIZER and ASSIGN message A-52
   without INITIALIZER procedure A-56

Volume, disc file: finding filenames 3-86

WAIT^FILE Procedure, SIO A-36
WRITE procedure
   completion 3-4
   example
      adding record to end of entry-sequenced 4-63
      adding record to end of relative 4-62
      record insertion, key-sequenced file 4-54
   syntax description 3-132
Write, security check 3-8

WRITEUPDATE procedure
   dependency on current position 2-10
   example
      random deletion with primary key 4-58
      random update, key-sequenced file 4-55
      sequential read by pri-key w/delete 4-59
      sequential read by pri-key w/update 4-57
   syntax description 3-136
WRITEUPDATEUNLOCK procedure
   completion 3-4
   dependency on current position 2-10
   syntax description 3-140
WRITE^FILE Procedure, SIO A-38
Writing records 1-23
   considerations for structured files 4-3
   with WRITE 3-132

$RECEIVE Handling with SIO A-62
   data transfer protocol A-62

READER'S COMMENTS

Tandem welcomes your feedback on the quality and usefulness of its publications. Please indicate a specific *section* and *page* number when commenting on any manual. Does this manual have the desired completeness and flow of organization? Are the examples clear and useful? Is it easily understood? Does it have obvious errors? Are helpful additions needed?

Title of manual(s): _____

FOLD ➤

FOLD ➤

FROM:

Name _____

Company _____

Address _____

City/State _____ Zip _____

A written response is requested, yes no ?

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA. U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

TANDEM
COMPUTERS, INC.

Attn: Technical Publications
19333 Vallco Parkway
Cupertino, CA, U.S.A. 95014

FOLD

STAPLE HERE

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, CA 95014