900 Series HP 3000 Computer Systems

Using KSAM XL

HP Part No. 32650-90168
Printed in U.S.A. 1994

Third Edition
E0394

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	April 1990	A.40.00
Second Edition	June 1992	B.40.00
Third Edition	March 1994	C.50.00

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s not based on the PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

In This Book

This manual provides programmers with descriptions and examples of the KSAM XL file format and its accessing routines. The material is organized into nine chapters and two appendixes.

The "Introduction" describes the KSAM XL file, its indexing mechanism, and its standard recovery methods.

"Creating a KSAM XL File" describes different methods of creating a KSAM XL file. Standard commands have been adapted to create and load a KSAM XL file. Intrinsics are also available to create and open a KSAM XL file. The key characteristics of the file are specified in command or intrinsic parameters.

"Obtaining File Information" describes the `LISTFILE` command and two intrinsics that access file and key characteristics of a KSAM XL file.

"Opening and Closing the File" describes the intrinsic opening and closing routines. Note that a KSAM XL file can also be created at the time the file is opened.

"Reading File Data" provides various methods of accessing records both sequentially and randomly using different intrinsics.

"Writing and Updating Record Data" provides the intrinsics that are used to write and append records to a file. File updates and deletions are also described.

"Protecting the File and Its Data" provides several methods of maintaining file integrity through error checking routines and regular file backups. Special information is provided for protecting data when access is shared. This section also describes recovering from system and software aborts and from internal file structure corruption.

"Migration and Mixed Mode Processing" offers migration strategies for transferring CM KSAM files to an MPE/iX system and to the KSAM XL file format.

"KSAM XL Intrinsics" provides all syntax and operation notes regarding the use of the KSAM intrinsics.

Two appendixes provide COBOL 68 intrinsics and BASIC/V intrinsics that may be needed for program maintenance. These intrinsics are not intended for use in new program development. They are provided here only as a maintenance aid for COBOL 68 or BASIC/V programs.

Conventions

`UPPERCASE` In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command        Command        COMMAND
```

It cannot, however, be entered as:

```
comm           com_mand       comamnd
```

italics In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

`COMMAND` *filename*

bold italics In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

`COMMAND(`***filename***`)`

punctuation In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

underlining Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

`{ }` In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`:

`COMMAND` { `ON` / `OFF` }

`[ ]` In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted:

`COMMAND` *filename* `[OPTION]`

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated.

`COMMAND` *filename* [`OPTION` / *parameter*]

Conventions (continued)

[...] In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

```
[,parameter][...]
```

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

```
[parameter][,...]
```

| ... | In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **A, AB, BA**, or **B**. The elements cannot be repeated.

```
{A}
{B}| ... |
```

... In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

Δ In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

```
(parameter)Δ(parameter)
```

key The symbol key indicates a key on the keyboard. For example, RETURN represents the carriage return key or Shift represents the shift key.

CTRL*character* CTRL*character* indicates a control character. For example, CTRL**Y** means that you press the Y key while holding down the control key.

Contents

Figures

Tables

1

Introduction

The Keyed Sequential Access Method (KSAM) is a method of organizing data records according to the content of key fields within the record. This method allows sequential processing of records without relying on the physical location of the record in the file.

Every record in a KSAM file contains a primary key field. The content of this field determines the logical sequence of each record. Alternate keys offer different sequences for accessing the same records.

KSAM XL is a KSAM file format that functions in the native mode (NM) environment of the MPE/iX operating system. It is a single file that consists of an index area that contains key indexes, and a data area that contains data records.

A primary key and up to fifteen alternate keys can be defined for a KSAM XL file. Key values are arranged in ascending order based on the data type of the field.

Note

The MPE V/E KSAM file format is also available on the MPE/iX system and is referred to as CM KSAM. It is a two-file format consisting of a data file and a key file. Refer to the *KSAM/3000 Reference Manual* (30000-90079) for a description of the format, file building instructions, and maintenance information.

KSAM XL File Format

A KSAM XL file is a single file consisting of an index portion and a data portion. Figure 1-1 provides a general representation of the contents of a KSAM XL file.

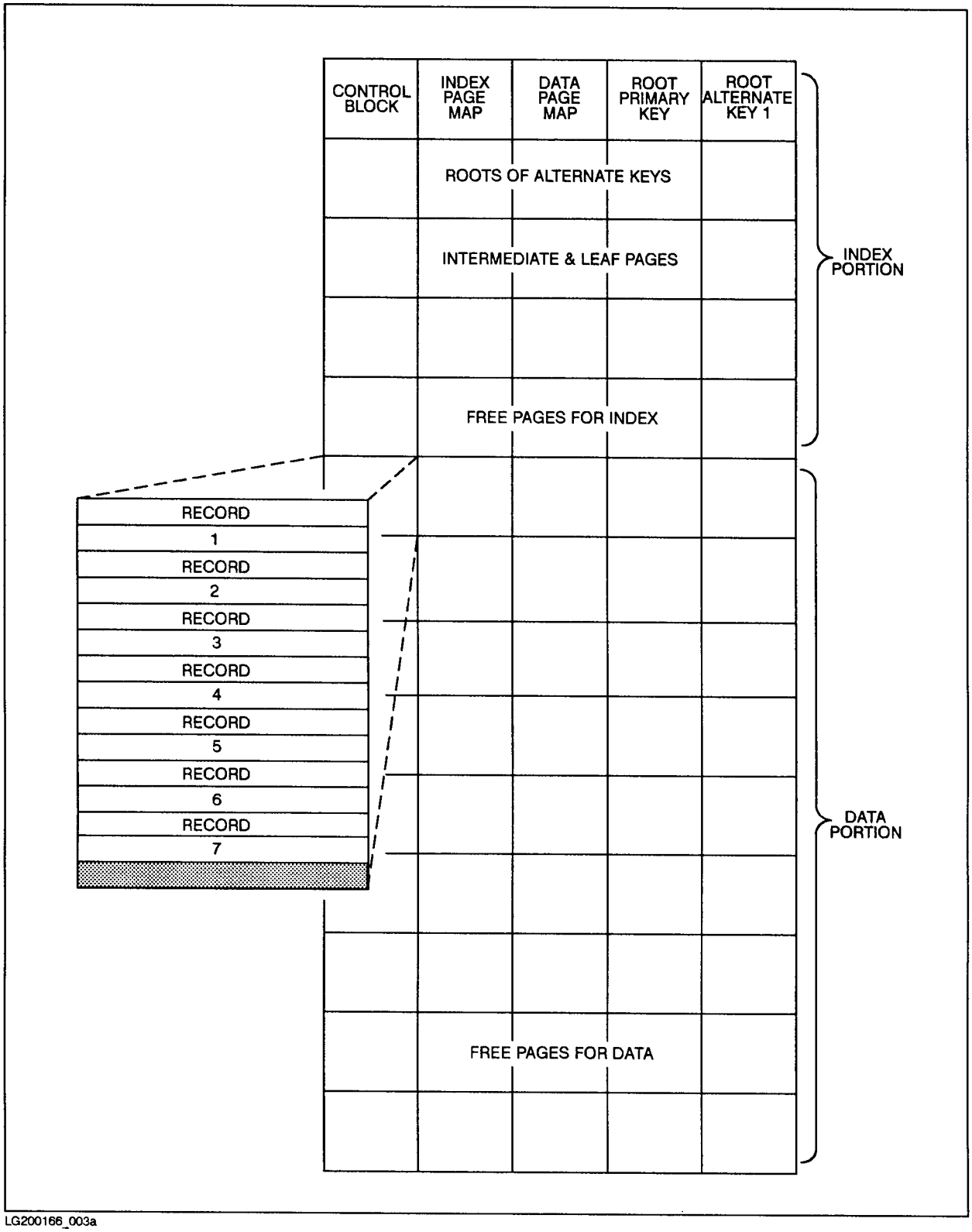


Figure 1-1. General Representation of the KSAM XL Format

Index Area

The index area contains a control block, bit mappings for the pages of the index and data areas, and the key indexes. The control block contains the file specifications and key specifications established when the file was built. It also contains pointers to the index and data page maps to manage the file's space.

A key index contains a key value and pointer for each record. This index data is arranged in ascending order based on the key value. If alternate keys are identified for the file, alternate indexes are created for each key.

When the file is opened for sequential processing, records can be accessed by physical location in the file or by key sequence. The selected key index supplies a pointer to the data record. Figure 1-2 shows how key index entries relate to the appropriate records in the file.

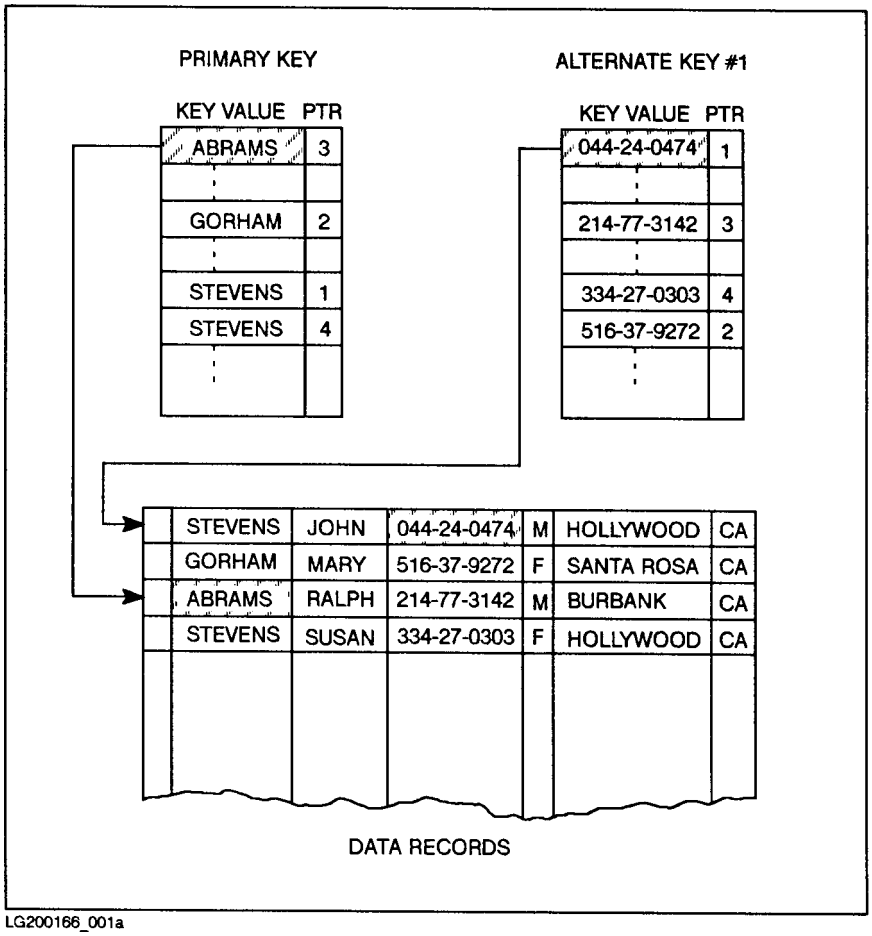


Figure 1-2. A Simplified View of the KSAM File Structure

The index portion of the file is organized in a tree structure. Figure 1-3 provides a diagram of a simple structure. The entry point of the structure, the root, either points to the location of an entry or directs the search to branches of the structure for higher or lower entries. The branches narrow the search, again, either to an entry location or to an ever-decreasing number of higher or lower entries. The lowest level, or leaves, provides pointers to the locations of the remaining records. Root, branch, and leaf pages for each key are contained in the index portion of the KSAM XL file.

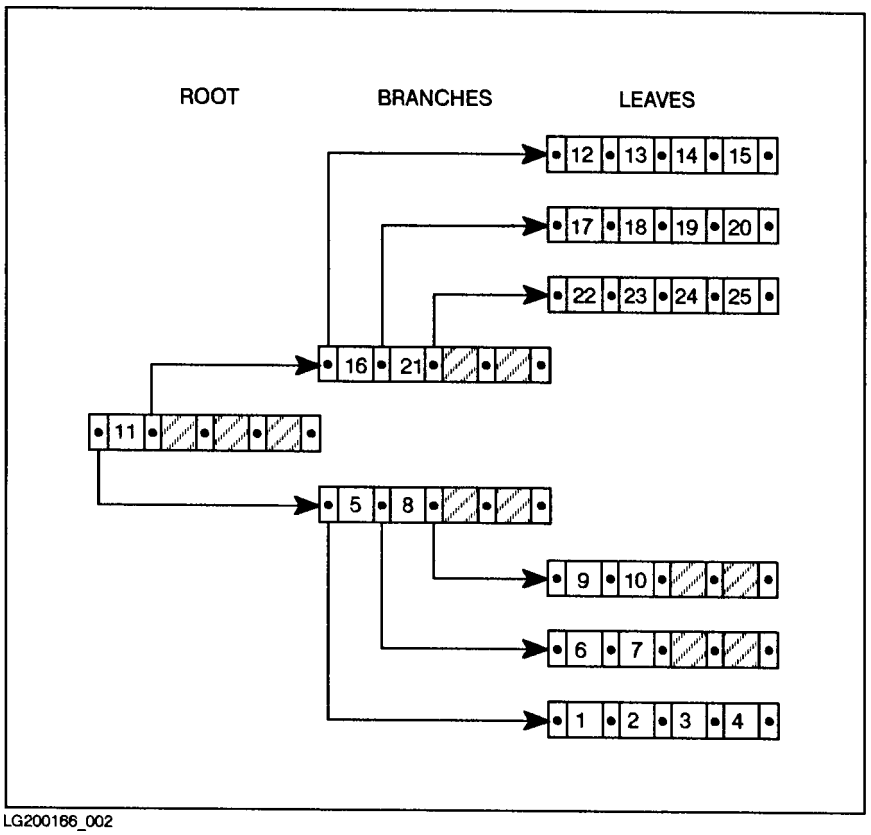


Figure 1-3. Simple Index Tree Structure

Data Area

The data area of the file follows the index area and contains all the data records. A 4-byte record header precedes each record. The first byte of this record header specifies whether the record has been deleted. When records are written to a KSAM XL file, the data record is written to the data area first. Keys are then inserted in the appropriate indexes using the data area location for creating pointers.

By default, records are stored in chronological order. When new records are appended, they are written at the end of the file, maintaining the chronological order. As records are deleted, the record space is not recovered and reused.

If the `REUSE` option is specified when the file is built, new records appended to the file are written in available space throughout the

file, thus interrupting the chronological sequence. In this case, physical location of a record does not represent the chronological order of written records.

Any alterations to the data area of the file, such as additions, modifications, or deletions, are immediately available to subsequent accesses by any process. The file system guarantees the order of concurrent data access.

Automatic Recovery

Automatic recovery maintains minimal data loss, data consistency, and recoverability from system software and hardware failures. This recovery is provided by the transaction management facility. If a failure occurs, all transactions in progress are backed out automatically when the system is restarted. No data or key inconsistencies result.

2

Creating a KSAM XL File

You can create a KSAM XL file in several different ways:

- Using the `BUILD` command. The file name and file characteristics are specified in the command parameters. The file can then be loaded with data by using the FCOPY subsystem to load existing file data or by directing program output to the file.
- Copying an existing file using the FCOPY subsystem. File characteristics can be defaulted to those of the existing file or modified by using a file equation.
- Using `HPFOPEN` or `FOPEN` intrinsic parameters from within an application program. The intrinsic call creates and opens the file. The program's output can then be written to the opened file.

Creating the File With the BUILD Command

The `BUILD` command parameters define standard file characteristics, such as the file and record lengths, and file, record, and data types. For KSAM XL files, you must also specify characteristics of each key field and special KSAM options. The following list offers the most common file characteristics that you need to decide before building a KSAM XL file.

- The file name.
- Size of the record.
- Record type of F for fixed-length records (required for KSAM XL files).
- Binary-coded or ASCII-coded data.
- Permanent or temporary file.
- Device class `DISC` (required for KSAM XL files).
- The maximum number of records.
- The language ID.
- A file type of `KSAMXL` (required for KSAM XL files).
- Information about each key (repeated up to sixteen times); at least one key is required :
  - Type of key data.
  - The location of the first byte of the key.
  - The length of the key.

- Random insertion or sequential insertion of the key, if duplication is allowed.

- Record numbering starting with 0 or 1.
- Reuse of deleted record space or no reuse.
- Specify default data block size or allow KSAM XL to select data block size.

KSAM XL File Characteristics

The key characteristics, the method of file numbering, and the reuse option are unique to KSAM XL files. Each key must be defined in the `BUILD` command's `;KEY=` parameter. Record numbering and the reuse option must be specified if the default values are not acceptable.

Key Characteristics

The `;KEY=` parameter of the `BUILD` command encloses all key characteristics in parentheses. Individual characteristics for a single key are separated by commas. Each key description is separated from the next by a semicolon. The following example shows a `;KEY=` parameter that defines two keys. Four characteristics are defined for each key: key type, location, size, and duplication method.

```
;KEY=(B,9,5,RDUP;I,17,3,DUP)
```

The following descriptions list the available options for the definition and use of keys. Four characteristics are defined for each key: key type, location, size, and duplication method.

The key type defines the data type of the key field. The type is identified by a keyword or its abbreviation. In the previous example, the first key field contains byte data and the second is an integer. The following list provides the valid key types.

byte or B	Byte data field.
integer or I	Integer data field.
real or R	Real number.
IEEE real or E	IEEE floating-point decimal number.
numeric or N	Numeric field.
packed or P	Packed decimal field, odd number of digits.
*packed or *	Packed decimal field, even number of digits.

The key's location is determined by the position of the first byte of the field in relation to the beginning of the record. The first byte of the record is considered to be 1. Only one key can start at a particular location. In the previous example, the first key begins in byte 9, the second in byte 17.

The size of the key must be specified in bytes. Specific use of any key is determined by its definition. The ranges listed below indicate the maximum possible values. The maximum length of the key varies by data type, as specified in the following list:

byte	1 to 255 bytes.
integer	1 to 255 bytes of integer data.
real	1 to 255 bytes of real number data.
IEEE real	4, 8, or 16 bytes of IEEE real number data.
numeric	1 to 28 bytes of numeric data.
packed	1 to 14 bytes of packed decimal data (odd number of characters).
*packed	2 to 14 bytes of signed packed decimal data (even number of characters).

The duplication key characteristic is an optional field. If a key must be unique, such as an account number or social security number, no additional parameters are made. The default value is no duplication. If the key can be duplicated, there are two methods of inserting duplicate key values in the index's duplicate key chain.

`DUP` specifies that each new duplicate key is inserted at the end of the duplicate key chain, maintaining chronological order.

`RDUP` specifies that each new duplicate key is inserted randomly in the duplicate key chain. `RDUP` is used if the reuse option is selected. With `RDUP`, chronological order is not maintained.

First Record Number

The `;FIRSTREC=` parameter of the `BUILD` command specifies the number of the first record in the file. Several record retrieval methods use record numbers to identify the physical location of a record. You can specify whether to use "0" or "1" to identify the first record. The default value is 0.

REUSE Option

KSAM XL files can reuse deleted record space if the `REUSE` option is specified. This option, however, increases the allocated space reserved for the file by 15 percent and distributes free space evenly throughout the file when the file is initially loaded. When a record is to be added to the file, free space is available so that a search for record space is not lengthy. When a record is deleted, its space is added to the free space available.

The `NOREUSE` option, the default value, does not allow the reuse of deleted record space. This option maintains physical record order. A new record is appended to the end of the file, even if other records have been deleted. If many records are added and deleted, the file continues to expand in size. In such cases, it is recommended that the file be copied regularly to eliminate the unusable space if disk space is needed.

Language ID

The optional `;LANG=` parameter of the `BUILD` command 224 specifies the native language specifies the native language of the data in the file. You can select the language by entering a code of up to three

digits or by entering the language name. To find out what languages can be accessed on your system, enter `RUN NLUTIL.PUB.SYS`. Any of the listed language IDs can be entered in this field. The default language is Native-3000. Different affectd languages may cause the sequential ordering of records to be affected.

OPTMBLK/DEFBLK Option

Users can assure efficient disk space utilization by using the `OPTMBLK` option of the `BUILD` command. When specified, `OPTMBLK` allows KSAM XL to choose the optimal data block size based on the record size of a file. Refer to *MPE/iX Commands Reference Manual Volumes 1 and 2* (32650-90003 and 32650-90364) for more information on using this option.

The `LISTFILE, 7` command displays the optimal data block size and the 8 bit value of the flagword of the KSAM parameter.

```
:BUILD XOPTM;KSAMXL;KEY=(B,1,4);OPTMBLK
:LISTFILE XOPTM,7

KEY     KEYTYPE      KEY LOCATION   KEY SIZE  DUP/RDUP
---     -------      ------------   --------  --------
 1     BYTE             1       4  NONE

NUM KSAM KEYS: 1               FIRST KSAM RECORD: 0
LANGUAGE     : ENGLISH         REUSE RECORD     : NO
VERSION      : 2               COMPUTE BLK SIZE : OPTMBLK
DATA         :
```

Figure 2-1. Creating a KSAM XL file using the `OPTMBLK` parameter

The `DEFBLK` parameter of the `BUILD` command allows the user to select a data block size of 4K bytes. If neither `OPTMBLK` nor `DEFBLK` is specified, the data block size defaults to `DEFBLK` (block size of 4K bytes).

```
:BUILD XDEF;KSAMXL;KEY=(B,1,4)
:LISTFILE XDEF,7

KEY     KEYTYPE      KEY LOCATION   KEY SIZE  DUP/RDUP
---     -------      ------------   --------  --------
 1     BYTE             1        4  NONE

NUM KSAM KEYS: 1               FIRST KSAM RECORD : 0
LANGUAGE     : ENGLISH            REUSE RECORD   : NO
VERSION      : 2               COMPUTE BLK SIZE  : DEFBLK
DATA         :
```

Figure 2-2. Creating a KSAM XL file with the data block size set at 4K bytes (default)

Use the FILE command along with the FCOPY command to copy a new KSAM XL file to one where the data block size is chosen using OPTMBLK.

Users with existing KSAM XL files of 4K bytes can convert their files by using FCOPY. Specify the OPTMBLK option in the file equation. This allows KSAM XL to select the data block size in the file equation. If a file equation does not specify either option, FCOPY uses the FROM= file's setting of OPTMBLK or DEFBLK.

Sample BUILD Command

Figure 2-3 builds a sample KSAM XL master file to process 80-byte accounts receivable records in English. The maximum size of the file is 100 records. Record numbering in the sample file begins with number 1. Reuse of deleted record space is allowed.

In this sample, four key fields are defined to sequence data for various programming functions:

- A unique 6-digit account number as the primary key.
- A 25-character field containing the client's last name.
- A 5-digit zip code field.
- A 3-character branch ID.

Figure 2-3 creates the ARMSTR file with the preceding specifications using the BUILD command. (Note that ampersands have been included at the end of each line to continue the command on subsequent lines to improve readability.)

```
:BUILD ARMSTR.MGR;REC=-80,,F,ASCII;&
DEV=DISC;DISC=100;KSAMXL;&
KEY=(N,4,6;&            Specifies account number (primary) key
B,10,25,RDUP;&          Defines the last name key
N,65,5,RDUP;&           Defines the zip code key
B,70,3,RDUP);&          Defines the branch ID key
FIRSTREC=1;REUSE ;LANG=5      Specifies that the first record is
                              identified by number 1,
                              that deleted record space can be reused,
                              and that the native language is English.
```

Figure 2-3. Building the AR Master File

Specifying an Indirect File

To reduce errors, the characteristics for key data fields can be contained in an indirect file and referred to in the BUILD command. Such a file can be created using an editor, such as HP EDIT. The information is structured as it would be if it were included in the command. The format of the key data in the indirect file is shown in the following example.

```
(N,4,6;&
B,10,25,RDUP;&
N,65,5,RDUP;&
B,70,3,RDUP)
```

Figure 2-4 shows the command for setting up the same accounts receivable master file as in Figure 2-3. The KEY= parameter, however, refers to the indirect file named KEYDATA for the key data specifications. The character ^ specifies that an indirect file contains the data.

```
:BUILD ARMSTR.MGR.AR;REC=-80,,F,ASCII;DEV=DISC;&
DISC=100;KSAMXL;KEY=^KEYDATA;&
FIRSTREC=1;REUSE ;LANG=5
```

Figure 2-4. Using a Key Data File

Loading Data to a KSAM XL File

Once the file has been created, you can load it with data from another file or from a program. The FCOPY subsystem is often used to load data from one file to another. Any type of file can be used as the input file for this process. FCOPY is executed by entering the subsystem name. It displays a prompt (>) while awaiting input.

```
:FCOPY
>
```

The FROM= command identifies the source file containing the data to be copied. The TO= parameter specifies the target file to which the data will be copied. The following example copies the existing master file records contained in OLDMSTR to the newly created KSAM XL file, ARMSTR.

```
>FROM=OLDMSTR.MGR.AR;TO=ARMSTR.MGR.AR
```

The FCOPY subsystem can also be used to copy a KSAM XL file's records in a different sequence. The KEY= parameter identifies the relative record location of the key to be used to establish the new sequence of records. The following example copies records from the old master file to the new file in alphabetical order by client name. The location of the client name field (10) is identified in the KEY= parameter.

```
>FROM=OLDMSTR.MGR.AR;TO=ARMSTR.MGR.AR;KEY=10
```

The FCOPY subsystem can create a new KSAM XL file if the source file is a KSAM XL file and if no file characteristics need to be changed. To identify the type of file to be built as a KSAM XL

file, the name is enclosed in parentheses. If the parentheses are not included, a standard file type is created.

The following example creates a new master file, duplicating the file and key specifications from the original file ARMSTR. Note that the file name is enclosed in parentheses, identifying the file type of the new file as a KSAM XL file type.

```
>FROM=ARMSTR.MGR.AR;TO=(ARMBACK.MGR.AR)
```

Modifying Existing File Specifications While Copying

A file equation can be used to modify file specifications of an existing file. The FCOPY subsystem can be used to copy data from an existing file into a new file using a back reference to the file equation for the new specifications. The following example copies data from the file DOC to a new KSAM XL file DOC1. The file type and key specifications for the new file are specified in the file equation.

```
FILE DOC1=DOC1;KSAMXL;KEY=(b,1,4)
FCOPY FROM=DOC;TO=*DOC1;NEW
```

Building a KSAM XL File Programmatically

The HPFOPEN and FOPEN intrinsics can be used within a program to create and open a KSAM XL file in a single step. As with the BUILD command, file and key characteristics are provided as parameter data.

Note

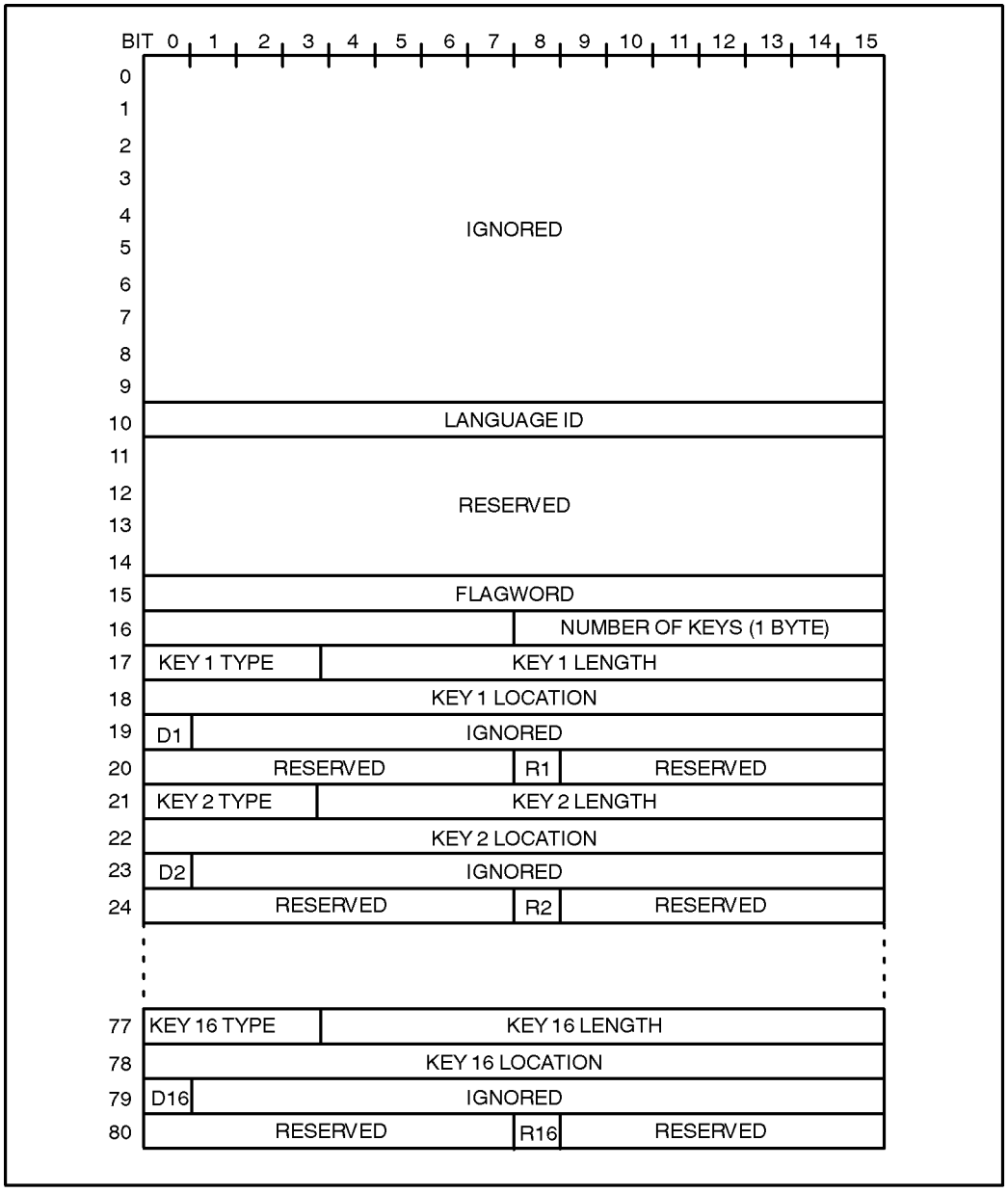
The HPFOPEN intrinsic can be used only in an MPE/iX environment. If a program is to be developed for both MPE/iX and MPE V/E systems, the FOPEN intrinsic should be used. Refer to "Mixed Mode Operation" in Chapter 8 for information regarding cross development.

The unique KSAM XL file and key characteristics are contained in an array that varies in length from 40 to 162 words. The format of the array is shown in Figure 2-5. Characteristics for a maximum of sixteen keys need to be specified in the array. Standard file characteristics are contained in the file options parameter of the intrinsic.

Language ID Enter the three digit code for the native language that you desire. To find out what languages can be accessed on your system, enter `RUN NLUTIL.PUB.SYS`. A list of languages and their IDs is displayed on the screen. Any of the listed language IDs can be entered in this field.

Flag word The flag word contains two bytes defining the KSAM XL file characteristics:

Bits	Value/Meaning
15:1	Reserved.
14:1	Enter a 1 if record numbering is to start with 1.
	Enter 0 if record numbering is to start with 0.
13:1	Enter 1 if only sequential writing by primary key is allowed.
	Enter 0 if random writing by primary key is allowed.
12:1	Enter 1 if deleted record space can be reused.
	Enter 0 if deleted record space cannot be used.
11:1	Enter 1 if a language type is specified.
	Enter 0 if a language type is not specified.
10:1	Enter 1 if the primary key cannot be changed with the `FUPDATE` intrinsic for files that are opened for sequential processing.
	Enter 0 if the primary key can be changed with the `FUPDATE` intrinsic for files that are opened for sequential processing.
9:1	Enter 1 if the file is programmatically accessed by the COBOL programming language. Enter 0 if the file is not programmatically accessed by the COBOL programming language. This enables KSAM to process COBOL information according to COBOL standards.
8:1	Enter 1 if KSAM XL is to select the optimal data block size. Enter 0 if KSAM XL is to use the default data block size.
0:9	Enter 0. These bits are reserved and must contain zeros.



LG200166_004

Figure 2-5. KSAM XL Parameter Format

Number of Keys

Enter a digit between 1 and 16 in word 16 to specify the number of keys to be defined for this file. Refer to Figure 2-5 for the location of this field.

Key Parameters

The following parameters are defined for each key. The information about each key is similar to the `BUILD` command's `KEY=` parameter.

key type

Enter one of the following codes specifying the type of data the key will contain.

Code	Key Data Type
1	Byte key (1 to 255 bytes)
2	Short integer key (255 bytes)
3	Integer key (255 bytes)
4	Real number key (255 bytes)
5	Long real number key (255 bytes)
6	Numeric display key (1 to 28 bytes)
7	Packed decimal key, odd number of digits (1 to 14 bytes)
8	Packed decimal key, even number of digits (2 to 14 bytes)
9	IEEE floating-point decimal key (4, 8, or 16 bytes)

key length

Enter the length of the key in bytes. A maximum of 255 bytes is allowed, but the length is dependent on the type of key data specified.

key location

Enter the relative location in bytes of the key field in the record. Note that the first byte of the record is considered 1.

duplicate key flag

Enter 1 if duplicate key values are allowed for this key.

Enter 0 if duplicate key values are not allowed for this key.

random insert flag

This field specifies the method of inserting duplicate key values. To use this feature, the previous duplicate key flag must be set to 1.

Enter 0 if duplicate key values are to be inserted at the end of the duplicate key chain.

Enter 1 if the duplicate key values are to be inserted randomly in the duplicate key chain.

Figure 2-6 provides an example of the declarations that are needed to define and load a KSAM XL parameter array using Pascal/iX.

Chapter 4 provides an example of an HPFOPEN intrinsic call that creates and opens a KSAM XL file.

```
type
  bit1=0..1;
  bit4=0..15;
  bit7=0..127;
  bit8=0..255;
  bit12=0..4095;
  bit15=0..32767;
  bit16=0..65535;
  pac80     = packed array [1..80] of char;
  ksam_rec    = packed record
         case integer of
         1 : (bitword : bit16);
         2 : (lang_id : bit16);
         3 : (resrvd0 : bit8;
            optm_blk: bit1;
            cm   : bit1;
            chg_primary : bit1;
            kslang : bit1;
            ksreuse : bit1;
            seq_random : bit1;
            rec_numbering : bit1;
            resrvd2 : bit1);
         4 : (resrvd3 : bit8;
            num_keys : bit8);
         5 : (key_type : bit4;
            key_length : bit12);
         6 : (dflag : bit1;
            maxkeyblk : bit15);
         7 : (resrvd5 : bit8;
            rflag : bit1;
            resrvd6 : bit7);
         8 : (key_location : bit16);
         end;
  ksam_struc   = ARRAY[0..80] OF ksam_rec;
  .
  .
  .
```

Figure 2-6. KSAM Parameter Settings

```
var
  ksam_param,
  ksamparam : ksam_struc;
  keylocation,
  reserved : bit16;
  :
begin
  ksamparam[10].lang_id := 5;
  ksamparam[16].resrvd3 := 0;
  ksamparam[16].num_keys := 1;
  ksamparam[17].key_type := 2;
  ksamparam[17].key_length := 5;
  keylocation := 5;
  ksamparam[18].bitword := keylocation;
:
```

KSAM Parameter Settings (continued)

The HPFOPEN intrinsic uses item number pairs to identify intrinsic parameters. Item number 54 is paired with the KSAM parameter array to define the KSAM XL key structure. Other item number pairs that relate to KSAM XL files specifically are listed below:

10 This item number identifies the KSAM XL file type. Enter 3 to indicate that a KSAM XL file is to be created.

17 A KSAM XL file can be accessed only as its own type. Enter 0 for a KSAM XL file.

The FOPEN intrinsic can also be used to create and open a KSAM XL file. The same KSAM parameter array is used as an FOPEN parameter option. The FOPEN intrinsic uses parameter values rather than item number pairs to identify file characteristics and the KSAM key value array. Refer to Chapter 4 for a description of the FOPEN intrinsic.

Using Related Commands

Several MPE/iX commands can be used for KSAM XL files. KSAM XL files can be deleted and renamed using the same commands used with standard files. File attributes can be modified with a file equation.

Deleting a KSAM XL File

KSAM XL files can be deleted using the PURGE command. As with standard files, the file named in the PURGE command is deleted. The accounts receivable file can be deleted using the following command.

```
PURGE ARMSTR.MGR.AR
```

Renaming a KSAM XL File

The RENAME command can be used to change the name of an existing KSAM XL file. The file name specified in the command is deleted. The parameters for the RENAME command are the same as for standard files. The file name specified in the command is deleted. The first file name is the current name of the KSAM XL file. The second file name is the new name of the file.

```
RENAME ARMSTR.MGR, OLDMSTR.MGR.
```

Modifying File Attributes

The FILE command declares the file attributes to be used when an existing file is opened. It can be used with KSAM XL files as well as standard files. The FILE command's keywords (;KSAMXL, ;KEY, ;FIRSTREC, ;LANG, ;REUSE, ;NOREUSE, ;OPTMBLK and ;DEFBLK) perform the same functions as they do for the BUILD command.

The FILE command can be used to override system default file specifications or specifications supplied with the HPFOPEN or FOPEN intrinsic. The new specifications remain in effect for the entire job or session unless they are revoked by the RESET command or superseded by another FILE command.

3

Obtaining File Information

You can obtain file information about an existing file using the LISTFILE command or the FGETINFO and FGETKEYINFO intrinsics. You can also add specific information about your file by writing it to a user label. The FWRITELABEL and FREADLABEL intrinsics provide access to user labels.

Displaying File and Key Information

Use the LISTFILE command to display the file specifications used to build the file. This command lists descriptions of one or more disk files at the level of detail you select. The level of display detail is controlled by the option number or keyword parameter following the file name.

A KSAM XL file does not have a unique file code. The file's structure can be discerned from a LISTFILE display using option 1 (SUMMARY) or 2 (DISC). When displayed in this manner, the character K is appended to the file type of a KSAM XL file to distinguish it from standard files. A file code of KSAM identifies a CM KSAM data file. A file code of KSAMK identifies a CM KSAM key file. The following example displays summary information for a KSAM XL file, a CM KSAM key file, a CM KSAM data file, and a standard file.

```
:LISTFILE,1
ACCOUNT=       AR            GROUP=        MGR

FILENAME   CODE   ------------LOGICAL RECORD-------
                    SIZE  TYP        EOF      LIMIT

ARMSTR              160B  FAK          0        115
EMPKEY     KSAMK    128W  FB        1742       1742
EMPLOYEE   KSAM     256B  FA           0       1023
CLIENT               80B  FA           1          1
```

Figure 3-1. File Type Display

Two options display the key specifications for a KSAM XL file. Option 5 (DATA) displays the file specifications and key data for the file. Option 7 (UNIQUE) displays information that is unique to the file type. For KSAM files, this displays the key data without the file specifications.

Figure 3-2 provides an example of the LISTFILE command using option 5 (DATA) and the display it generates.

```
:LISTFILE ARMSTR.MGR.AR,5
**************************
FILE: ARMSTR.MGR.AR

FILE CODE : 0                    FOPTIONS: ASCII,FIXED,NOCCTL,KSAMXL
BLK FACTOR: 1                    CREATOR : **
REC SIZE: 160(BYTES)             LOCKWORD: **
BLK SIZE: 160(BYTES)             SECURITY--READ    : ANY
EXT SIZE: 0(SECT)                          WRITE   : ANY
NUM REC:  0                                APPEND  : ANY
NUM SEC:  2160                             LOCK    : ANY
NUM EXT:  2                                EXECUTE : ANY
MAX REC:  115                           **SECURITY IS ON
                                 FLAGS   : n/a
NUM LABELS: 0                    CREATED : MON, NOV 13, 1989,  3:35 PM
MAX LABELS: 0                    MODIFIED: MON, NOV 13, 1989,  3:35 PM
DISC DEV #: 16                   ACCESSED: MON, NOV 13, 1989, 10:15 PM
CLASS     : DISC                 LABEL ADDR: **
SEC OFFSET: 0

KEY         KEY TYPE        KEY LOCATION   KEY SIZE   DUP\RDUP

  1         NUMERIC                4          6   NONE
  2         BYTE                  10         25   RDUP
  3         NUMERIC               65          5   RDUP
  4         BYTE                  70          3   RDUP

NUM KSAM KEYS: 4                          FIRST KSAM RECORD: 1
LANGUAGE     : ENGLISH                    REUSE RECORDS    : YES
PRIMARY KEY  : RANDOM                     COBOL            : NO
VERSION      : 2                          COMPUTEBLK SIZE  : OPTMBLK
```

Figure 3-2. File Information Display

For a KSAM file, the file specifications, as well as the key information specified when the file was built, is displayed. (Note that the keyword DATA could have replaced the option number 5 in the LISTFILE request in the preceding example.) This display could be abbreviated to display only the key data by using option 7 (UNIQUE) as shown in Figure 3-3.

```
:LISTFILE ARMSTR.MGR.AR,7
*************************
FILE: ARMSTR.MGR.AR

KEY          KEY TYPE         KEY LOCATION    KEY SIZE    DUP\RDUP

  1          NUMERIC                     4           6    NONE
  2          BYTE                       10          25    RDUP
  3          NUMERIC                    65           5    RDUP
  4          BYTE                       70           3    RDUP

NUM KSAM KEYS: 4                          FIRST KSAM RECORD: 1
LANGUAGE     : ENGLISH                    REUSE RECORDS    : YES
PRIMARY KEY  : RANDOM                     COMPUTE BLK SIZE : OPTMBLK
VERSION      : 2
```

Figure 3-3. Key Information Display

Accessing File Information from a Program

The FGETINFO intrinsic obtains a file's access and status information based on the parameters identified in the intrinsic call. Embedded parameters that are not desired are indicated by commas. Parameters omitted from the end of the list do not need to be indicated.

In the following example, the intrinsic call returns the end of file in the variable named LSTREC. This number represents the physical number of the last record in the file if the REUSE option has not been specified. This variable can be used to position a pointer to read the last physical record with the FREADC or FREADDIR intrinsic.

```
FGETINFO(FILENO,,,,,,,,,LSTREC);
```

The FGETINFO intrinsic returns the following file information.

- The fully qualified file name.
- The foptions specified in the format of the FOPEN intrinsic.
- The aoptions specified in the format of the FOPEN intrinsic.
- The logical record size associated with the file.
- The type and subtype of the device being used for the file.
- The logical device number associated with the device on which the file resides.
- The hardware address of the device.
- The data file code.
- The current physical record pointer setting.

- The number of logical records currently in the data file.
- The number of the last logical record that could be contained by the file.
- The total number of logical records passed to and from the user during the current access of the file.
- The block size of the file.
- The disk extent size associated with the file.
- The maximum number of disk extents allowed for the file.
- The number of user labels allowed for the file.
- The name of the user who created the file.
- The sector address of the label of the file.

Accessing Key Information From a Program

Like the `FGETINFO` intrinsic, the `FGETKEYINFO` intrinsic provides access and status information about the keys of a KSAM file. It provides detailed information about the key location, type, and length in a parameter format similar to the `FOPEN` intrinsic key parameter. The `FGETKEYINFO` intrinsic also provides access information, such as a count of the number of times the key file has been accessed by various intrinsics, or the date and time the file was created, closed, updated, or written to.

Accessing User-Defined Labels

A user label is an optional method of adding documentation to your file. You can write your own labels to a KSAM file with the `FWRITELABEL` intrinsic. For example, you can use a label to enter the date and time of the last file update. These labels are read with the `FREADLABEL` intrinsic.

Specify the number of user labels to be created in the *userlabel* parameter of the `FOPEN` intrinsic. In order to write labels, the file must be open. To do so, set the *aoptions* parameter of the `FOPEN` intrinsic to one of the write, input/output, or update access specifications.

The following example shows the intrinsic call to write information to the second file label.

```
FWRITELABEL(KFILNUM,LABELBUF,60,1);
```

In this example, the 60 halfwords of text contained in the variable LABELBUF are to be written in the second user label. Note that label numbering starts with zero. The second label is identified by the number 1 in the last parameter. If this parameter contains zero or is omitted, the first label is written.

You can read the contents of user labels using the FREADLABEL intrinsic. During the normal reading of a file, user labels are skipped. The FREADLABEL intrinsic, therefore, should be called immediately after the file has been opened. To read a user label, the file must be opened with read, input/output, or update access, and the user labels to be read must be identified.

Issue the following FREADLABEL intrinsic call to read the user label written in the previous example.

```
FREADLABEL(KFILNUM,LABEL2,,1)
```

The variable LABEL2 returns the contents of the second user label. By default, the call returns 128 halfwords from the label.

4

Opening and Closing the File

Some application programming languages offer commands for opening and closing KSAM files (for example, the `ORGANIZATION IS INDEXED` clause in COBOL). If not, use the `HPFOPEN` or `FOPEN` intrinsic to open the file, and the `FCLOSE` intrinsic to close the file. See the appropriate application language reference manual for details on how to call intrinsics.

Opening an Existing KSAM XL File

The `HPFOPEN` and `FOPEN` intrinsics both open KSAM XL files, as well as other file types. `HPFOPEN` is designed to be more flexible and offers more options than the `FOPEN` intrinsic. `HPFOPEN`, however, can be used only in an MPE/iX environment. If the program is to be used in both MPE/iX and MPE V/E environments, use the `FOPEN` intrinsic.

Using the HPFOPEN Intrinsic

The `HPFOPEN` intrinsic uses pairs of item numbers and items for optional parameter passing. An *itemnum* parameter passes an integer by value to define the parameter and expected data type of the value passed in its corresponding *item* parameter.

To open an existing permanent file, file characteristics do not have to be specified. This information is obtained by the file management system from the file's label.

Most often, the item number pairs that are needed to open an existing KSAM XL file include the file designator, its domain, and access options. The domain identifies the location of the file to be opened. The access option defines the method of access allowable for the file. In some cases, the dynamic locking option and exclusive option need to be specified if more than one process is to access the file.

Figure 4-1 provides a portion of a Pascal program that calls the `HPFOPEN` intrinsic to open the accounts receivable KSAM XL file. It presents the *itemnum* and *item* definitions and declarations as well as the `HPFOPEN` intrinsic call. In the example, the file is opened for update access, allowing all intrinsic usage. It also allows dynamic locking and shared access for concurrent use with other processes.

```
procedure open_permanent_KSAM_file;

const
   formal_designator_option = 2;
   domain_option            = 3;
   access_type_option       = 11;
   dynamic_locking_option   = 12;
   exclusive_option         = 13;
   ASCII_binary_option      = 53;

type
   pac160    = packed array [1..160] of char;

var
   file_num  : integer;
   status    : integer;
   file_name : pac160;
   permanent : integer;
   update    : integer;
   lockable  : integer;
   shared    : integer;
   ascii     : integer;

begin
   file_num        := 0;
   status          := 0;
   file_name       := '%ARMSTR.MGR.AR%';
   permanent       := 1;
   update          := 5;
   lockable        := 1;
   shared          := 3;
   ascii           := 1;

   HPFOPEN(file_num, status,
          formal_designator_option, file_name,
          domain_option, permanent,
          access_type_option, update,
          dynamic_locking_option, lockable,
          exclusive_option, shared,
          ASCII_binary_option, ascii
          );

   if status <> 0 then handle_file_error (file_num, status);
end;
```

Figure 4-1. Opening an Existing KSAM XL File with HPFOPEN

The *file_num* parameter is used to return a file number to the calling program. This file number is used to identify the file in subsequent intrinsic calls. The *status* parameter returns a numeric code identifying the success or failure of the file opening process.

For clarity, the *itemnum* parameters in the previous example have been defined as constants. This is not necessary for intrinsic use. The following `HPFOPEN` intrinsic call provides the same options as the preceding example, but the *itemnum* parameters are identified by number. Note that the corresponding *item* parameters are variables that contain the appropriate selections. These variables would have to be defined and declared as in the previous sample.

```
HPFOPEN(file_num, status,
       2, file_name,
       3, permanent,
       11, update,
       12, lockable,
       13, shared,
       53, ascii
       )
```

Using the FOPEN Intrinsic

Only the file designator and the domain need to be specified to open an existing file with the `FOPEN` intrinsic. Rather than the *itemnum/item* pairs in `HPFOPEN`, the `FOPEN` intrinsic parameters are specified as bit groupings. The domain must be specified in the *foption* parameter (bits 14:2). The *aoption* parameter must be set if an access other than read needs to be specified.

The `FOPEN` intrinsic uses positional parameters to specify options. This means that the sequence of parameter data defines the parameter to which it refers. For example, in an `FOPEN` intrinsic call, the file designator is followed by the *foption* parameter, which is followed by the *aoption* parameter. The following example shows the `FOPEN` intrinsic call to open an existing KSAM XL file for read only access:

```
file_num:=FOPEN(file_name,3)
```

The variable *file_num* returns the file number for use in subsequent intrinsic calls. The *foption* value 3 specifies that an existing user file is to be opened (bits 14:2= (binary) 11). Because no *aoption* parameter was specified, the file is opened with read only access, the default.

To open an existing file with update access, specify the access mode in the *aoption* parameter. The other parameters remain the same. The following example opens the file with update access.

```
file_num:=FOPEN(file_name,3,5)
```

In this example, the *aoption* value 5 specifies update access for the file (bits 12:4 = (binary) 0101). This level of access allows all other intrinsic calls for this file. Other binary access selections include:

binary 0000 or 0 To read the file.

binary 0001 or 1 To write to the file for the first time.

binary 0010 or 2 To append records to the file.

binary 0100 or 4 To allow both read and write access.

binary 0101 or 5 To update records in the file.

If your file requires shared access and you are accessing records using pointer-dependent procedures, you must allow dynamic locking in the file opening procedure and use the `FLOCK` and `FUNLOCK` intrinsics to protect your transactions from access by another process. This ensures that no other user changes or deletes the record after you have positioned the pointer to it. In this case, the *aoption* parameter must be set to allow both shared access and dynamic locking, as well as to specify the access method. Note that the *aoption* parameter can be entered in octal notation listing "%" instead of "binary". This allows setting the shared and dynamic locking bits.

```
FILENUM:=FOPEN(FILNAME,3,OCTAL ('340')
```

The preceding example allows shared access (bits 8:2 = binary 11) and dynamic locking (bits 10:3=1) with read only access (bits 12:4=0) .

Opening a New File

As discussed in Chapter 2, a file can be created when it is opened using the `HPFOPEN` or `FOPEN` intrinsics. The file characteristics must be specified, as well as the formal file designator, the domain, and the access method. The most common item numbers used to create and open KSAM XL files with the `HPFOPEN` intrinsic include:

2 The file designator.

10 A file type of 3 for KSAM XL files.

11 An access option of 1 for writing records to a new file.

19 The record length.

35 The maximum file length.

50 Either a disposition of 2 for a temporary file or 1 for a permanent file.

53 ASCII or binary record data.

54 The KSAM key parameter defining primary and alternate key descriptions.

Figure 4-2 presents a portion of a program that builds and opens a KSAM XL file.

```
type
   bit1=0..1;
   bit4=0..15;
   bit7=0..127;
   bit8=0..255;
   bit12=0..4095;
   bit15=0..32767;
   bit16=0..65535;
   pac80  = packed array [1..80] of char;
   ksam_rec = packed record
              case integer of
              1 : (bitword : bit16);
              2 :  (lang_id : bit16);
              3 : (resrvd0 : bit8;
                   select_blk_size;
                   cm      : bit1;
                   chg_primary : bit1;
                   kslang  : bit1;
                   ksreuse : bit1;
                   seq_random : bit1;
                   rec_numbering : bit1;
                   resrvd2 : bit1);
              4 : (resrvd3 : bit8;
                   num_keys : bit8);
              5 : (key_type : bit4;
                   key_length : bit12);
              6 : (dflag : bit1;
                   maxkeyblk : bit15);
              7 : (resrvd5 : bit8;
                   rflag : bit1;
                   resrvd6 : bit7);
              8 : (key_location : bit16);
              end;
   ksam_struc = ARRAY[0..80] OF ksam_rec;
:
var
   file_num  : integer;
   status    : integer;
   file_name      : pac80;
   ksam_type      : integer;
   write_access   : integer;
   line_len       : integer;
   file_len       : integer;
   save_perm      : integer;
   ascii          : integer;
   ksamparam      : ksam_struc;
   keylocation,
   reserved       : bit16;
```

Figure 4-2. Opening a New KSAM XL File with HPFOPEN

```
:
begin
   file_num        := 0;
   status          := 0;
   file_name       := '%ARMSTR.MGR.AR%';
   ksam_type       := 3;
   write_access    := 1;
   rec_len         := 80;
   file_len        := 100;
   save_perm       := 1;
   ascii           := 1;
:
 ksamparam[10].lang_id  := 5;
ksamparam[16].resrvd3  := 0;
ksamparam[16].num_keys := 1;
ksamparam[17].key_type := 2;
ksamparam[17].key_length := 5;
keylocation := 5;
ksamparam[18].bitword := keylocation;
:
HPFOPEN(file_num, status,
       2, file_name,
       10, ksam_type
       11, write_access
       19, rec_len,
       35, file_len
       50, save_perm,
       53, ascii
       54, ksamparam
       );

   if status <> 0 then handle_file_error (file_num, status);
end;
```

Opening a New KSAM XL File with HPFOPEN (continued)

To create a new KSAM XL file using the FOPEN intrinsic, file characteristics and KSAM key information are specified in the positional parameters. In most cases, the *foption*, *aoption*, *recsize*, *ksamparam*, and *filesize* parameters must be specified. Commas identify those positional parameters for which the default specifications are used. Figure 4-3 provides an FOPEN intrinsic call that creates a KSAM XL file with write access to build the file.

```
type
   bit1=0..1;
   bit4=0..15;
   bit7=0..127;
   bit8=0..255;
   bit12=0..4095;
   bit15=0..32767;
   bit16=0..65535;
   pac80          = packed array [1..80] of char;
   ksam_rec       = packed record
                      case integer of
                        1 : (bitword : bit16);
                        2 :  (lang_id : bit16);
                        3 : (resrvd0 : bit8;
                             select_blk_size;
                             cm      : bit1;
                             chg_primary : bit1;
                             kslang  : bit1;
                             ksreuse : bit1;
                             seq_random : bit1;
                             rec_numbering : bit1;
                             resrvd2 : bit1);
                        4 : (resrvd3 : bit8;
                             num_keys : bit8);
                        5 : (key_type : bit4;
                             key_length : bit12);
                        6 : (dflag : bit1;
                             maxkeyblk : bit15);
                        7 : (resrvd5 : bit8;
                             rflag : bit1;
                             resrvd6 : bit7);
                        8 : (key_location : bit16);
                      end;
   ksam_struc     = ARRAY[0..80] OF ksam_rec;
var
   file_num       : integer;
   file_name      : pac80;
   ksamparam      : ksam_struc;
   keylocation    : bit16;
```

Figure 4-3. Opening a New KSAM XL File with FOPEN

```
begin
   file_num                := 0;
   file_name               := 'ARMSTR.MGR.AR ';
   ksamparam[10].lang_id   := 5;
   ksamparam[16].resrvd3   := 0;
   ksamparam[16].num_keys  := 1;
   ksamparam[17].key_type  := 2;
   ksamparam[17].key_length := 5;
   keylocation             := 5;
   ksamparam[18].bitword   := keylocation;
   file_num:=FOPEN(file_name,6148,1,-80,,ksamparam,,,,100)
end;
```

Opening a New KSAM XL File with FOPEN (continued)

Closing a KSAM XL File

The FCLOSE intrinsic terminates access to a file. The disposition and the security code parameters control the file's retention and its authorized users. When closing an existing file, you usually close it with both parameters set to zero.

```
FCLOSE(FILNUM,0,0)
```

You cannot change an existing permanent file to a temporary file using the FCLOSE intrinsic. A temporary file, however, can be closed as a permanent file by specifying the domain in the disposition field. To close a newly created temporary file, set the disposition parameter (bits 13:3) to 1 to save it as a permanent file, or 2 or 3 to keep it as a temporary file. Note that the disk space bit of the disposition parameter (bits 11:2) should not be used for a KSAM XL file.

```
FCLOSE(FILNUM,1,0)
```

The security code parameter (*seccode*) specifies the level of access security assigned to the file. It is set only for a permanent file. A value of 1 gives you exclusive access to the file; 0 allows access by other users. Regardless of the value assigned to the *seccode* parameter when closing an existing file, the type of security applied to the file when it was created is maintained.

In the following example, a new file is closed and saved as a permanent file in the system file domain (*disposition* = 1), and access to the file is restricted to the file's creator (*seccode* = 1).

```
FCLOSE(FILENUM,1,1)
```

5

Reading File Data

KSAM files offer multiple record retrieval options using primary and alternate keys, and logical and physical record numbers. The following list identifies the methods of reading KSAM file data:

- Sequential access:
  - By primary key.
  - By alternate key.
  - In physical record order.
- Random access:
  - By key value.
  - By logical record number.
  - By approximate key match.
  - By partial key.
  - By physical record number.

KSAM XL uses two types of pointers to identify the location of records to be read: the logical record pointer and the physical record pointer. The logical record pointer points to a key in the index, which points to a data record. This pointer is used to locate records by key. The physical record pointer points directly to a data record. This pointer is used to locate records by their physical location in the file.

Intrinsics that use pointers are either pointer-dependent or pointer-independent. Pointer-dependent intrinsics expect the pointer to be positioned in order to execute correctly. Pointer-independent intrinsics execute regardless of where the pointer is positioned.

KSAM XL maintains an advance flag to specify whether or not to advance the pointers before the specific function. If the flag is set to TRUE, pointers are advanced before performing the intrinsic function. If the flag is set to FALSE, the intrinsic function is performed without advancing the pointers first.

Intrinsics have been developed to position pointers and to read records in sequence or randomly, by key value and by record number. Table 5-1 identifies the intrinsics used to access files and identifies those pointers that are set by each.

Table 5-1.
Pointer and Advance Flag Settings for Reading

Intrinsic	Reads Advance Flag	Sets Pointer	Sets Advance Flag	Pointer Dependant
FFINDBYKEY	no	both	no	no
FFINDN	no	both	no	no
FPOINT	no	both	no	no
FREAD	yes	both	yes	yes
FREADBYKEY	no	both	no	no
FREADC	yes	PHYS	yes	yes
FREADDIR	no	PHYS	yes	no
FSPACE	yes	both	no	yes

Note

COBOL II and Business BASIC provide KSAM file access routines that read records by key value. Refer to your programming language manual for details.

Sequential Access by Primary Key

Many processes retrieve records in a sequence, to systematically perform a function on each record. The primary key sequence is usually used for such routines. The file opening routine (an HPFOPEN or FOPEN intrinsic call) prepares for the most common record retrieval method by positioning the pointers at the record containing the lowest value of the primary key. A call to the FREAD intrinsic, after the file is opened, reads the first record in the primary key sequence.

After reading the first record, the logical record pointer remains in the same position. The next FREAD repositions the logical pointer as well as the physical record pointer to the next sequential record in ascending key sequence and reads the record. Although FREAD may position both pointers, it uses the logical date pointer to locate the particular record. An end-of-data condition occurs when the last logical record is passed. At this point, the CCG condition code is set and returned to your process.

Sequential Access by Primary and Alternate Key

Two intrinsics, FFINDN and FFINDBYKEY, can be used to set the logical pointer to the lowest value of an alternate key field. The FFINDN intrinsic identifies the first record by using a logical record number. The FFINDBYKEY intrinsic uses a key value to determine the first record.

When the first record has been located, the FREAD intrinsic reads the first record specified by the alternate key. Subsequent reads reposition the logical pointer and read the next logical record.

The FREADBYKEY intrinsic can also be used to position the logical pointer by alternate key value. In this case, however, the user must know the lowest value of the alternate key. An approximate value cannot be used with this intrinsic.

Specifying the Record Number

The FFINDN intrinsic positions the pointer to the record specified by the logical record number of the appropriate key. To position the pointer to the particular record of a key, the intrinsic parameters identify the particular key of interest and then the record number.

Depending on how the file was built, the first record of any key is identified by 1 or 0. Use option 5 or 7 of the LISTFILE command to determine how records are numbered in the file you are accessing. A negative record number also positions the pointer to the lowest value in the key field.

The key location identifies the key field to be used. Again, use option 5 or 7 of the LISTFILE command to determine the location of the desired key (ffn_key_location). The following example identifies the record of an alternate key and reads the specified record:

```
  FFINDN(filenum,ffn_rec_number,ffn_key_location);
  .
  .
  .
  lgth :=FREAD(filenum,fr_record,fr_tcount);
```

Figure 5-1. FFINDN Intrinsic Sample

Specifying a Key Value

The FFINDBYKEY intrinsic can also be used to position the pointer to an alternate key. This intrinsic is intended to position the pointer to the first occurrence of a record value that matches or is greater than the key value. This is referred to as an approximate match. To position the pointer to the first record of the key, supply a key value that is less than any value of the key and specify a relational operator of 1 (greater than) or 2 (equal to or greater than). For example, a relational operator of 1 locates the first record having a key value greater than the key value provided.

Figure 5-2 sets the pointer to the lowest value of the alternate key by searching for the first occurrence of a key value greater than (relop = 1) the value "0000":

```
  fby_keyvalue   := '0000';
  fby_keylocation := 1;
  fby_keylength  := 4;
  fby_relop    := 1;
:
  FFINDBYKEY(filenum,fby_keyvalue,fby_keylocation,fby_keylength,fby_relop);
:
  lgth := FREAD(filenum,fr_record,fr_tcount);
```

Figure 5-2. FFINDBYKEY Intrinsic Sample

Sequential Access by Partial Key Value

The FFINDBYKEY intrinsic can be used to point to those records that contain a common portion of a key field. The intrinsic parameters (*key value*, *key length*, and *relational operator*) identify the partial value to be matched, the number of characters to be compared in the key field, and whether the record should equal the value or be greater than the value.

Only the common portion of the key is specified in the *key value* field. For example, to list all records with a zip code beginning with 943 but ending in any combination of numbers, 943 is entered in the *key value* field.

The *key length* parameter identifies the portion of the key field to be used in the comparison. For example, to list all records with a zip code beginning with 943, a *key length* of 3 would be specified. This means that only the first three characters of the five-character field are used in the comparison.

The *relational operator* limits the operation to only those records that meet the criteria. The relational operators that can be specified are 0 (equal to), 1 (greater than), and 2 (equal to or greater than). Figure 5-3 searches for the first occurrence of a record containing a partial key of "M0".

```
  fby_keyvalue   := 'M0';
  fby_keylocation := 1;
  fby_keylength  := 2;
  fby_relop    := 0;
:
  FFINDBYKEY(filenum,fby_keyvalue,fby_keylocation,fby_keylength,fby_relop);
:
  lgth := FREAD(filenum,fr_record,fr_tcount);
```

Figure 5-3. Partial Key Search Sample

To read all records containing "M0", a series of `freads` would be issued and a comparison made in the program to see when the key field *did not* contain "M0" or the end of the file reached.

Random Access of a Single Record

A record can be accessed randomly by a particular key value or by its relative or physical record number.

Using a Key Value

The `FREADBYKEY` intrinsic is recommended for retrieving records randomly. The desired *key value* and the *key location* are specified in the intrinsic parameters. The index of the specified key is checked for a matching key value and the appropriate record is read.

If an exact key value match is not found, an error condition is returned. Because of this, the `FREADBYKEY` intrinsic is not appropriate when searching for an approximate key value or the lowest value of a key. Use the `FFINDBYKEY` intrinsic in such cases.

```
target   := '    ';
tcount   := -8;
keyvalue  := '15 ';
keylocation := 5;

lgth :=FREADBYKEY(filenum,target,tcount,keyvalue,keylocation);
```

Figure 5-4. Accessing a Record by Key Value

Using the Relative Record Number

Records can also be accessed randomly using the `FFINDN` intrinsic. To use this intrinsic, however, you need to know the record's relative record number in its key sequence.

Using a Physical Record Number

The `FREADDIR` intrinsic reads a single record based on its physical record number in the file. The record number is supplied as parameter data in the intrinsic call. Record numbering starts with either 1 or 0, depending on the specifications made when the file was built.

The `FPOINT` and `FREADC` intrinsics can be used to read a record based on its physical record number. The `FPOINT` intrinsic positions the pointers to the record identified by its physical record number in the file. The `FREADC` intrinsic is then used to read the record based on the physical record pointer without reference to the record's index location.

In this case, the `FREAD` intrinsic could also be used to read the record, because the `FPOINT` intrinsic also sets the logical record pointer to the record that it located by physical record number. By default, the key used is the primary key for that record. An alternate key is

used, however, if such a key was specified by a previous call to the FFINDBYKEY or FREADBYKEY intrinsic.

Note

This is true for the reads on the previous examples of FFINDN FFINDBYKEY, FREADBYKEY intrinsics that sets the key of reference for succeeding reads.

Sequential Access in Physical Record Order

A sequential access in physical record order is really a series of random accesses by physical record number. The FPOINT and FREADC intrinsics are used to read records in order of their physical location in the file. The FPOINT intrinsic sets the physical record pointer to the position specified in its record number parameter. The FREADC intrinsic reads the record specified by the physical record pointer without reference to the logical record pointer. A subsequent FREADC intrinsic advances the physical record pointer to the next physical record. Any record containing a delete flag is ignored and is not read.

The FREADDIR intrinsic also reads files in physical record order. It positions the pointer to the record specified in the record number parameter. A subsequent FREADDIR intrinsic call repositions the physical record pointer to the next physical record. Note that deleted records are not ignored with this intrinsic. It is recommended, therefore, that you use the FPOINT and FREADC intrinsics to read records sequentially in physical record order. Use the FREADDIR intrinsic only to read a single record identified by its physical record number.

The FGETINFO intrinsic returns the physical record pointer setting, as well as other information, for the record most recently accessed. This number is returned in the record pointer parameter and can be used in a subsequent FPOINT or FREADDIR intrinsic call.

Shared File Access

If only one process is accessing a file, setting a pointer and reading a record in a two-step process does not present a problem. Shared file access, however, presents potential retrieval contention. If a pointer is positioned to retrieve a particular record by one process, another process could modify or delete the record before the original process reads it. The FLOCK and FUNLOCK intrinsics should be used to ensure proper record retrieval in any program that allows shared access to its file.

Note

File locking keeps the file inaccessible to other users until the file is unlocked. This could be a potential source of performance problems. A different file structure may be more suitable for applications in a shared environment, such as IMAGE/3000, etc.

An FLOCK intrinsic call should be made prior to a pointer positioning and record reading procedure to ensure that the proper retrieval is executed. The FUNLOCK intrinsic restores shared access once the retrieval is completed. Once the file is unlocked, do not assume that the pointer is still valid. Before using the pointer again, reposition it. The following sequence shows the appropriate locking procedure to ensure the proper sequence of records.

```
FLOCK
FFINDBYKEY (sets the logical pointer)
  FREAD loop (reads records in key sequence)
FUNLOCK
```

6

Writing and Updating Record Data

When records are written to a file for the first time, they are usually written sequentially. Following execution of an `FWRITE` intrinsic, the logical record pointer is positioned at the next sequential record in key sequence or at the end-of-file marker if the record is the last in sequence.

Updating and deleting records also rely on pointer positioning. The logical and physical record pointers are usually positioned by a read procedure, as discussed in Chapter 5. Typically, a read procedure precedes an update or delete procedure to verify that the correct record has been found. Table 6-1 specifies the advance flag and pointer usage of each of the writing, updating, and deletion intrinsics.

Table 6-1. Pointer and Advance Flag Settings for Writing

Intrinsic	Reads Advance Flag	Sets Pointer	Sets Advance Flag	Pointer Dependant
`FREMOVE`	no	both	no	yes
`FUPDATE` (keys unchanged)	no	none	yes	yes
`FUPDATE` (keys changed)	no	both	no	yes
`FWRITE`	no	both	no	yes

Writing New Records

The FWRITE intrinsic writes new records to a new or existing file from a buffer in your program. Index entries for primary and alternate keys are entered automatically for each record written.

Depending on how the file was created, records may be written in random or sequential order. If the REUSE option is specified, each record is written to the next available space. If the NOREUSE option is specified, all records are written at the end of the file.

Records written to an existing file either overwrite existing records or are appended to existing records. This is determined by the access option of the *aoptions* parameter, selected in the HPFOPEN or FOPEN intrinsic call.

Following each write procedure, the logical record pointer is positioned at the next sequential record in key sequence or at the end-of-file marker. When the physical bounds of either the data area or index area of the file is reached, a CCG condition code is returned to your program.

Note that the control parameter of the FWRITE intrinsic must be included in the intrinsic call for compatibility. It has no meaning for KSAM XL files.

When writing records to a file that has shared access, file locking should be used. The HPFOPEN or FOPEN intrinsic call must allow dynamic locking. An FLOCK intrinsic should be included before pointers are positioned and records are written. Unlock the file using the FUNLOCK intrinsic when the write procedure is complete.

Updating Existing Records

To update a record in a KSAM XL file, the HPFOPEN or FOPEN intrinsic call to open the file must specify update access. This is set by the *aoption* parameter. Normally, you would read the record with one of the read intrinsics , to verify its contents before modification.

The FUPDATE intrinsic writes the contents of the buffer area over the contents of the last record accessed. This buffer area is identified in an FUPDATE intrinsic parameter. The written record must contain all the key values expected by the file. If only a portion of the record is updated, specified by the *tcount* parameter, this portion must contain all primary and alternate key values. If it does not, a CCL condition is returned and the update does not take place.

Deleting a Record

The intrinsic FREMOVE effectively removes the current record from the KSAM XL file. When executed, the 4-byte record header is modified, identifying the record as deleted. All key entries pointing to this record are deleted from the indexes. Although the data still occupies record space in the file, it is no longer possible to access the record through standard read operations. Note that if deleted record space can be reused, this area can be overwritten by a new record.

The FREMOVE intrinsic checks only the logical record pointer, not the physical record pointer, to locate the record to be deleted. To delete a record located by its physical record pointer, precede the call to the FREMOVE intrinsic with the FPOINT intrinsic. The FPOINT intrinsic locates the record by its physical record pointer but sets both the logical and physical record pointers.

Note

If you use the FREADDIR or FREADC intrinsic to locate the record, only the physical record pointer is set. You may delete the wrong record because the logical record pointer was not set by the read procedure.

Shared Access

If access to the file is shared with other processes, any of these intrinsics should be preceded by FLOCK and FUNLOCK intrinsics. This controls access to the records and reduces contention while a modification procedure is being performed. All pointer positioning, read intrinsics, and writing, updating, and deletion procedures should be bounded by the FLOCK and FUNLOCK intrinsics to guarantee that the proper record is updated or deleted.

Note

File locking keeps the file inaccessible to other users until the file is unlocked. This could be a potential source of performance problems. A different file structure may be more suitable for applications in a shared environment.

7

Protecting the File and Its Data

Attention must be paid to protecting a KSAM XL file's data. Check an intrinsic's status after a call to find information about a failed routine. The FCHECK and FERRMSG intrinsics provide error codes and messages after an intrinsic call has failed.

Various intrinsics control file access when a file is shared by more than one process. Locking and unlocking the file controls access to a shared file during critical modification operations.

The item numbers 2 and 6 of the FCONTROL intrinsic ensure that data is written to the disk before processing is allowed to continue. This protects the data from system and software aborts that may occur between the time that data is written to the transaction log and the time that it is actually written to the disk. Transaction management provides automatic recovery from system and software aborts.

Regular maintenance and file backups are needed for data protection against hardware failures or improper processing. If index corruption exists, files can be restored quickly through the FCOPY facility.

Checking Error Information

When a file intrinsic returns a condition code indicating that a physical input or output error has occurred, additional details can be obtained by calling the FCHECK intrinsic. The parameters of the FCHECK intrinsic can be designated to return the following error information:

- The error code that identifies the type of error that occurred.
- The transmission log value that specifies the number of words not read or written before the input or output error.
- The relative number of the block involved with the error.
- The number of logical records that were in the bad block at the time of the error.

This error information can be expanded to include a description of the error by calling the FERRMSG intrinsic. This intrinsic uses the error code returned by the FCHECK intrinsic. By supplying the returned FCHECK error code and defining a message buffer in the FERRMSG intrinsic call , a corresponding message can be displayed from your program. The error code returned by FCHECK and its

corresponding message can also be found in the *MPE/iX Intrinsics Reference Manual* (32650-90028).

Protecting Data When File Access is Shared

If a KSAM XL file is shared with another process, you need to ensure that the most current data and key index information is retrieved. Locking files controls other processes from accessing the file while a modification routine is processing. Such a modification routine should include the pointer positioning and reading routines that are associated with the modification routine. The `FUNLOCK` intrinsic allows the file to be shared again, once modifications are complete.

In a shared environment, it is recommended that you lock and unlock the file for pointer-related activities, such as `FREAD` or `FUPDATE` intrinsics using `FFINDBYKEY` or `FFINDN` intrinsics to locate the proper record.

Note

File locking keeps the file inaccessible to other users for an indeterminate length of time. This could be a potential source of performance problems. A different file structure may be more suitable for applications in a shared environment.

The following example shows how modification routines can be locked effectively by the placement of the `FLOCK` and `FUNLOCK` intrinsics.

```
FLOCK
FREADBYKEY
FUPDATE
FUNLOCK


FLOCK
FFINDBYKEY
FREAD loop
FUNLOCK
```

In many interactive processes, it is inefficient to keep a file locked while a user retrieves a record, decides whether it needs to be updated, makes appropriate changes, and writes the new record. In such cases, a simple read could retrieve the record's contents for the online user to see. Once a decision has been made to modify the contents, a new retrieval redisplays the record for updating. By rereading the file, the program will be able to verify that the correct record has been retrieved without locking the file for an excessive amount of time.

```
FLOCK
FREADBYKEY
FUNLOCK
⋮
 Other users can access and modify this record while
 the user decides how to update it.
⋮
FLOCK
FREADBYKEY
FUPDATE
FUNLOCK
```

Writing Directly to Disk

The FCONTROL intrinsic's *controlcode* parameter settings identify the control operation desired. A setting of 2 ensures that the requested output has been physically completed. (If the file is shared, you must lock the file before calling the FCONTROL intrinsic with a control code of 2.) A control code of 6 provides a similar function. It ensures that the requested output has been physically completed and that the end-of-file has been written.

Recovering from a System or Software Abort

File recovery after a system or software abort is provided automatically through transaction management. After a file has been created with the BUILD command or has been created and loaded using the HPFOPEN or FOPEN intrinsics, it is attached to system logging. If processing of a transaction is interrupted prior to its logical completion, the transaction is rolled back before processing is allowed to continue. A transaction is rolled back in the following cases:

- A system abort occurs.
- A process with an active logical transaction aborts.
- A transaction aborts.
- A transaction causes a deadlock condition.

If a KSAM XL file is created and loaded using FCOPY's NEW option, or an HPFOPEN or FOPEN intrinsic call, transaction logging is not attached until the file is closed. This provides a fast load mode that loads the file more quickly than if transaction logging was invoked. An abort during this load process, however, is not logged. If an abort occurs when creating and loading a file with FCOPY's NEW option or with the HPFOPEN or FOPEN intrinsic, restart the file loading process.

To protect initial loading, use the BUILD command to create the file. The file is attached to transaction management when the BUILD command is used. A file can also be attached manually by creating and loading the file with the HPFOPEN intrinsic and specifying the DOMAIN=CREATE option. With this option, the file is attached and system logging begins with the first access.

Backing Up KSAM XL Files

A regularly scheduled backup of all files is always advisable. The STORE/RESTORE facility used for most other files is also appropriate for backing up KSAM XL files to tape. The following commands provide a backup routine for a KSAM XL file.

```
FILE T=ARBACK;DEV=TAPE
STORE ARMSTR.MGR.AR;*T
```

Note

Do not use the TRANSPORT option of the STORE command with KSAM XL files. The TRANSPORT option is intended as a migration option for storing files from MPE/iX to MPE V/E systems.

Use the following commands to restore the file from tape:

```
FILE T=ARBACK;DEV=TAPE
RESTORE *T;ARMSTR.MGR.AR;KEEP;DEV=DISC;SHOW
```

The FCOPY utility can also be used to back up KSAM XL files on disk instead of tape. This allows a quick recovery with little delay. If sufficient resources are available, this is an effective and rapid method of backing up files.

Recovering from Index Corruption

If the file management subsystem detects file corruption, it does not allow writing, updating, and deletion activities. The file manager attempts to honor read requests, but the attempt may not be successful.

If index entries have been corrupted, create a new KSAM XL file using the BUILD command. When the file is built, load the data from the original file using the FCOPY utility with the KEY=0 option. The KEY=0 option does not access the indexes in the source file. It merely transfers data records from source to target, creating new index entries after each record is copied.

The following routine creates a new file and loads it with the data records from the original accounts receivable file.

```
:BUILD ARMSTR.MGR.AR;REC=-80,,F,ASCII;DEV=DISC;&
DISC=100;KSAMXL;KEY=(N,4,6;&
B,10,25,RDUP;&
N,65,5,RDUP;&
B,70,3,RDUP;&
FIRSTREC=1;REUSE

:FCOPY
>FROM=OLDMSTR.MGR.AR;TO=(ARMSTR.MGR.AR);KEY=0
>EXIT
```

Figure 7-1. Index Corruption Recovery

8

Migration and Mixed Mode Processing

MPE/iX offers two KSAM file formats: CM KSAM and KSAM XL. CM KSAM is the two-file KSAM structure used on MPE V/E systems.

KSAM XL, a single-file KSAM structure, is used only on MPE/iX systems. A KSAM XL file offers a more convenient single-file format.

Programs running in CM or NM can access either type of KSAM file. Use the FCOPY utility to migrate data and rebuild indexes from one KSAM file format to another.

Note

RPG Programmers:

Record-level locking cannot be used for either type of KSAM file on MPE/iX.

Similarities in KSAM File Features

Both file formats allow multiple keys to access data records and duplicate key values for specified keys. You can access records by various keys using constructs within the programming language. You can also use KSAM intrinsics to access records in various sequences.

Record retrieval can be by direct match of specific key value, by generic (or partial) key value, or by approximate match. Access of data records by physical record location may or may not match the primary key sequence, depending upon the order in which records were initially loaded.

Differences in KSAM File Features

Unlike CM KSAM files, KSAM XL data records and indexes are combined in a single file. The file limit of KSAM XL files is substantially larger than CM KSAM files. The physical size of the KSAM file is the same as the MPE/iX native mode flat file.

KSAM XL files allow only fixed-length records. CM KSAM files allow fixed-length or variable-length records. When the data is copied from CM KSAM variable-length records to KSAM XL fixed-length records, shorter records are padded with a fill character to the defined fixed-length record size. The fill character is specified during the file creation. The default fill character for an ASCII file is a blank. The default fill character for a binary file is a binary zero.

Both types of KSAM files allow the reuse of index entry space for deleted entries, but only KSAM XL allows the reuse of deleted record space. If chronological order of the records is not necessary, deleted record space can be reused.

`KSAMUTIL`, the utility used to create, rename, and purge CM KSAM files, does not support KSAM XL files. Instead, `KSAMUTIL` functions have been integrated into the following CI commands:

- `BUILD`
- `PURGE`
- `RENAME`
- `LISTFILE`

The FCOPY utility provides a method of migrating CM KSAM files to KSAM XL. KSAM XL files, however, cannot use the `NOKSAM` option in file copying.

Transaction management guarantees consistency and recoverability from system crashes. System logging provides this recoverability. System logging is attached after the first `FCLOSE` of the file. This occurs automatically with the `BUILD` command. Files built with `HPFOPEN` or `FOPEN` intrinsics are attached after the first `FCLOSE` intrinsic call or with the `DOMAIN=CREATE` option of the `HPFOPEN` intrinsic.

Migrating KSAM Files

The data records from an existing KSAM file on an MPE V/E system can be migrated to an existing KSAM XL file on an MPE/iX system. Perform the following steps to migrate an existing CM KSAM file with fixed-length records to a new KSAM XL file:

1. Store both the CM KSAM key file and data file to tape using the `TRANSPORT` option (used only if migrating to an MPE V/E system).
2. Restore both files to the MPE/iX machine (used only if migrating from an MPE V/E system).
3. Create the new KSAM XL file using the `BUILD` command.
4. Run the FCOPY utility.
5. Enter the appropriate `FROM=` and `TO=` parameters to copy the CM KSAM file to a KSAM XL file.
6. Exit FCOPY.
7. Delete the original data file and key file from the MPE/iX machine.
8. Rename the new KSAM XL file to the original CM KSAM data file name.

Note

KSAM XL files require fixed-length records. If the source CM KSAM file contains variable-length records, define the record length of the target file as the maximum length of the source records. When copying the file, FCOPY pads the source record with a fill character to create the target record size. The fill character is specified during the file creation. The default fill character for an ASCII file is a blank. The default fill character for a binary file is a binary zero.

The following entries show the FCOPY commands needed to migrate the CM KSAM file named `ARMSTR.MGR.AR` to an existing KSAM XL file. Note that in this example, the KSAM XL file structure already exists. You can create the file with the `BUILD` command or with the `FOPEN` or `HPFOPEN` intrinsics.

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=ARMSTRXL.MGR.AR
>EXIT
:PURGE ARMSTR.MGR.AR
:PURGE ARKEY.MGR.AR
:RENAME ARMSTRXL.MGR.AR, ARMSTR.MGR.AR
```

If record-level locking has not been used and no other migration issues exist, the source program can be run in compatibility mode. The program successfully accesses the new `ARMSTR` file. Refer to the

Migration Process Guide (30367-90007) for details about migrating application programs.

You can create a new KSAM XL file and copy the CM KSAM record data in a single step. Enclose the new file name in parentheses to specify that this is a KSAM XL file. If the KSAM XL file does not exist, a new file is created. A new file is also created by using the `NEW` option.

If you create the file and copy data to it using one command, however, you are not able to change the key structure. This would not be acceptable when copying variable-length records because the record length and record type parameters must be modified to acceptable values.

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=(ARMSTRXL.MGR.AR)
>EXIT
```

or

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=(ARMSTRXL.MGR.AR);NEW
>EXIT
```

`FCOPY` copies data records from the source file in the sequence identified by the primary key. Use the `KEY=` option to select a different sequence for copying the records. To retain the physical layout of the source file, specify `KEY=0`. This specification copies the records in the order that they reside in the source file without regard to a key.

Note

The `NOKSAM` option is not allowed with KSAM XL files.

Mixed Mode Operation

Application programs running in CM or NM can access either CM KSAM or KSAM XL files. If you are using an RPG application, do not specify any record locking features. RPG will deafult to file-level locking. This is especially important for cross-development for multiple environments.

In some organizations, cross development is necessary because satellite offices operate different types of systems. CM KSAM files can be used on both MPE V/E and MPE/iX systems. The KSAM XL file format can be used only on MPE/iX systems.

KSAM files can be copied from one type to another using the FCOPY utility. For detailed information on using the FCOPY utility, refer to the *FCOPY Reference Manual* (32212-90003).

To create a new CM KSAM file and copy data to it from an existing CM KSAM file , remember to identify both the data file and the key file for the target CM KSAM file. Use this method to back up current files or to create test files on an MPE V/E system. This process is described in detail in the *KSAM/3000 Reference Manual* (30000-90079).

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=(ARBACK.MGR.AR,ARBKEY.MGR.AR)
>EXIT
```

To create a new KSAM XL file and copy data to it from a CM KSAM file, specify only a single file name in the `TO=` parameter. (KSAM XL files include indexes and data records in a single file.) Enclose the new file name in parentheses to indicate that it is to be a KSAM XL file. The `;NEW` parameter is optional. Use this method to migrate files from an MPE V/E system to an MPE/iX system.

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=(ARMSTRXL.MGR.AR)
>EXIT
```

or

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=(ARMSTRXL.MGR.AR);NEW
```

To copy from one KSAM XL file to another existing KSAM XL file, enter a single file name for the target file. (KSAM XL files include indexes and data records in a single file.) Use this type of copy to back up current KSAM XL files or to create a test file on an MPE/iX system.

```
:FCOPY
>FROM=ARMSTR.MGR.AR;TO=ARBACK.MGR.AR
>EXIT
```

To create a new CM KSAM file and copy data to it from an existing KSAM XL file, remember that both the target data file name and the target key file name must be specified. Use this type of copy for cross-development.

```
:FCOPY
>FROM=ARMSTRXL.MGR.AR;TO=(ARDATA.MGR.AR,ARKEY.MGR.AR)
>EXIT
```

9

KSAM XL Intrinsics

The following section provides syntax and parameter definitions for the KSAM XL intrinsics. For details regarding status usage and data types, refer to the *MPE/iX Error Message Manual Volumes 1, 2 and 3* (32650-90066, 32650-90152, and 32650-90368) and the *MPE/iX Intrinsics Reference Manual* (32650-90028).

FCHECK

Returns specific details about error conditions that occurred when a file system intrinsic returned a condition code indicating an I/O error. `FCHECK` applies to files on any device.

Syntax

```
          I16V       I16       I16       I32      I16
FCHECK(filenum,fserrorcode,translog,blocknum,numrecs);
```

Parameters

filenum **16-bit signed integer by value (optional)**

Specifies the file number of the file for which error information is to be returned. If *filenum* is not specified or set to zero, error information is returned about the last failed `FOPEN` call.

fserrorcode **16-bit signed integer by reference (optional)**

Returns a file system error code indicating the type of error that occurred.

translog **16-bit signed integer by reference (optional)**

Returns the number of halfwords read or written if an I/O error occurred. (This value is recorded in the transmission log.)

blocknum **32-bit signed integer by reference (optional)**

Returns the physical record count for a nonspoolfile or the logical record count for a spoolfile:

- For fixed-length and undefined-length record files, the physical count is the number of physical records transferred to or from the file since `FOPEN`.
- For variable-length record files, the physical count is the last rewind, rewind/unload, space forward or backward to tape mark.

numrecs **16-bit signed integer by reference (optional)**

Returns the number of logical records in the bad block (blocking factor).

Operation Notes

`FCHECK` is used to determine the error conditions of the last failed `FOPEN` intrinsic call (even if a file number was not returned) by setting the *filenum* parameter to zero. In this case, only *fserrorcode* returns valid information.

Do not use `FCHECK` to determine error conditions of a last failed `HPFOPEN` call; error conditions are returned in the `HPFOPEN` *status* parameter.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The file number passed by *filenum* is invalid, or a bounds violation occurred while processing this request (*fserrorcode*=73).

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FCLOSE

Terminates access to a file on any device.

Syntax

```
            I16V     I16V       I16V
FCLOSE(filenum,disposition,securitycode);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be closed.

disposition **16-bit signed integer by value (required)**

Passes the disposition of the file, significant only for files on disk and magnetic tape.

Note

This *disposition* can be overridden by a corresponding parameter in a `FILE` command entered prior to program execution.

The *disposition* options are:

Bits	Value/Meaning
13:3	Domain disposition:
	000 No change. The *disposition* remains as it was before the file was opened. If the file is new, it is deleted by `FCLOSE`; otherwise, the file is assigned to the domain it belonged to previously. An unlabeled tape file is rewound and a labeled tape is rewound and unloaded.
	001 Close as a permanent file. If the file is a disk file, it is saved in the system file domain. A new or old temporary file on disk has an entry created for it in the system file directory. If a file of the same name already exists in the directory, an error code is returned and the file remains open. If the file is a permanent file on disk, this domain disposition has no effect.
	010 Close as a temporary job file (rewound). The file is retained in your temporary (job/session) file domain and can be requested by any process within your job/session. If the file is a disk file, the file name is checked. If a file of the same name already exists in the

temporary file domain, an error code is returned and the file remains open.

011 Close as a temporary job file (not rewound). This option has the same effect as domain disposition 010, except that tape files are not rewound.

100 Release the file. The file is deleted from the system.

101 Makes a permanent standard disk file temporary (valid only for standard disk files with either fixed-length, variable-length, or undefined-length record formats). The file is removed from the permanent file directory and inserted into the TEMPORARY file directory. (PM capability is required for this option.)

11:2 Disk space disposition (valid only for standard disk files with either fixed-length, undefined-length, or variable-length record formats):

00 Does not return any disk space allocated beyond the end-of-file marker.

01 Returns any disk space allocated beyond the end-of-file (EOF) marker to the system. The EOF becomes the file limit; records cannot be added to the file beyond the EOF.

10 Returns any disk space allocated beyond the end-of-file (EOF) marker to the system. The file limit remains the same; records can be added to the file beyond EOF, up to the file limit. The disk space disposition takes effect on each `FCLOSE`.

0:11 Reserved for MPE/iX.

securitycode **16-bit signed integer by value (required)**

Returns the type of security initially applied to the file (significant for new permanent files only). The valid options are:

Value	Meaning
0	Unrestricted access; can be accessed by any user, unless prohibited.
1	Private file creator security; can be accessed only by the creator.

Operation Notes

`FCLOSE` deletes buffers and control blocks where the process accessed the file. It also deallocates the device where the file resides, and it can change the *disposition* of the file. If `FCLOSE` calls are not issued for all files opened by the process, the calls are issued automatically by MPE/iX when the process terminates.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The file was not closed; an incorrect *filenum* was specified, or another file with the same name and *disposition* exists.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FCONTROL

Performs various control operations on a file or on the device where the file resides, including:

- Verifying I/O.
- Reading the hardware status word for the device where the file resides.
- Setting a terminal's timeout interval.
- Repositioning a file at its beginning.
- Writing an end-of-file marker.

Syntax

```
                I16V    I16V   *
FCONTROL(filenum,itemnum,item);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file for which the control operation is to be performed.

itemnum **32-bit signed integer by value (required)**

Specifies which operation is to be performed. (Refer to Table 9-1.)

item **type varies (required)**

Passes/returns a value associated with a control operation as indicated by the corresponding *itemnum* parameter. (Refer to Table 9-1.)

This parameter is ignored, but must be specified to satisfy internal requirements.

Table 9-1. FCONTROL Itemnum/Item Values

Itemmum	Mnemonic	Item Description
0	U16	General device control: The value specified is passed to the appropriate device driver. A value from the driver is returned in *item*. Not valid for spooled device files. Not applicable to KSAM files.
1	U16	Carriage control (CCTL): Not applicable to KSAM files.
2	I16	Complete I/O: Ensures that requested I/O has been physically completed. Valid only for buffered files. Posts the block being written (full or not). *Item* is ignored. A checkpoint record is written. In the event of a system crash, recovery is done to this state of the files.
3	U16	Device status: Returns a record containing information about the state of the device associated with the file immediately after the last I/O operation (including `HPFOPEN/FOPEN`) on the file. The record size and contents are device-dependent. Not applicable to KSAM files.
4	U16	Set timeout interval: Passes the timeout interval, in seconds, to be applied to input from the specified file. The maximum value allowed is 655.35 seconds. If input is requested from a file but is not received in this interval, the `FREAD` request terminates prematurely with CCL. The interval is specified in seconds and returned in *item*. If this interval is zero, any previously established interval is cancelled, and no timeout occurs. A timeout value should be used for programs reading from an unattended device to prevent "hangs". Timeouts can be used to terminate binary reads, but only as a safeguard to prevent a program from waiting too long for a read to complete. Only valid for terminal and message files. Only affects the next read if the addressed file is being read from the terminal; it must be reissued for each read. If this code is applied to a message file, *item* specifies the length of time that a process waits when reading from an empty file or writing to a full one and the timeout remains enabled until it is explicitly cancelled. Denotes a halfword in the stack that contains the time-out interval, in seconds, to be applied to input from the terminal. During block mode reads, the timer halts when a DC2 character is received. The block mode read timer is activated by the system software; these values are not user changeable. Not applicable to KSAM files.
5	U16	Reposition file at its beginning: The file is repositioned to the first logical record, the record with the lowest value in the current key.

Table 9-1. FCONTROL Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
6	U16	Write end-of-file: Marks the end-of-file (EOF) on disk. It performs the function of *itemnum*=2 and writes the file label. This guarantees that the end-of-file is correct and the extent bit map is updated. *Item* is ignored.
7	U16	Space forward to tape mark: Not used for KSAM XL files. For CM KSAM files, it clears the key and data buffers of all information and reads the first two sectors of the key file from disk to buffer.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FERRMSG

Returns a message corresponding to an FCHECK error number and enables error messages to be displayed from a program.

Syntax

```
                I16          CA        I16
FERRMSG(fserrorcode,msgbuffer,msglength);
```

Parameters

fserrorcode **16-bit signed integer by reference (required)**

Passes an error code returned by the FCHECK intrinsic, indicating which message to return in *msgbuffer*.

msgbuffer **character array (required)**

Returns the error message identified with *fserrorcode*. To contain the longest possible message, *msgbuffer* must be >= 72 bytes long.

msglength **16-bit signed integer by reference (required)**

Returns the length of the error message in *msgbuffer*. The length is returned in positive bytes.

Condition Codes

CCE	Request granted.
CCG	Request denied. No error message exists for this *fserrorcode*.
CCL	Request denied. The *msgbuffer* address was out of bounds, *msgbuffer* was not large enough, or *msglength* was out of bounds.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FFILEINFO

Returns information about a file.

Syntax

```
                I16V        I16V    *
FFILEINFO(filenum[,itemnum,item] [...]);
```

Note

Up to five *itemnum/item* pairs can be specified.

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file for which information is requested.

itemnum **16-bit signed integer by value (optional)**

Specifies which *item* value is to be returned. (Refer to Table 9-2.)

item **type varies (optional)**

Returns the value of the item specified in the corresponding *itemnum*. (Refer to Table 9-2.)

Table 9-2. FFILEINFO Itemnum/Item Values

Itemmum	Item Type	Item Description
1	CA	File designator (28 bytes): Returns the file designator of the file being referenced in the format: *filename.groupname.accountname* Must be >=28 bytes in length. Unused bytes are filled with right-justified blanks and a nameless file returns an empty string. The fully qualified name of the file referenced by *filenum* is returned as the value of this *itemnum*. Only names which can be expressed using MPE-only semantics are returned by this *itemnum*. If the name of the object referenced by *filenum* can not be expressed using MPE-name semantics a CCL condition code is returned. Calling `FCHECK` for *filenum* after this error occurs will result in error.
2	U16	File options: Returns file characteristics (refer to the *FFfoption* figure). The record format extension bit is returned as the *foption* (1:1) bit. Byte stream record format is represented as a record format extension of one with a variable record format *foption* (8:2) bits equal to 01. Directories, symbolic links, device links, pipes and FIFO's can not be represented by *foptions*. If the object referenced by *filenum* is one of these objects, a CCL condition code is returned. Calling `FCHECK` for *filenum* after this error occurs will result in error.
3	U16	Access options: Returns file access information (refer to the *FFaoption* figure).
4	I16	(CM) Record size: Returns the logical record size associated with the file: ■ If the file was created as a binary file, this value is positive and is in halfwords. ■ If the file was created as an ASCII file, this value is negative and is in bytes. For message files, when there is call to `FCONTROL` with *controlcode*=46, the value returned is the size of the data records, including the 4 byte header. Maintained for compatibility with MPE V/E-based systems only. CM record sizes are imposed when `FGETINFO` returns record size information on all file types. If the record size exceeds the limits, a zero is returned. Note: If a zero is returned, use item 67.
5	I16	Device type/subtype: Returns the type and subtype of the device being used for a KSAM, RIO, circular, or message file, or devices such as a tape drive, printer, or terminal where bits (0:8) indicate the device subtype, and bits (8:8) indicate the device type. If the file is not spooled or is opened as a spoolfile through the logical device, the actual value is returned. If an output file is spooled and was opened by device class name, the type and subtype of the first device in its class is returned. (This may be different from the device actually used.)

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
6	U16	Logical device number: Returns the logical device number of the device where the disk file label resides. ■ If the file is a disk file, the LDEV is the location of the file label. (File data can reside on the same device as the file label.) ■ If the file is spooled, the LDEV is a virtual device number that does not correspond to the system configuration I/O device list. ■ If the file is located on a remote computer, linked by a DS point-to-point or X.25 link, the left eight bits (0:8) are the LDEV of the distributed system (DS) device. ■ If the file is located on a remote computer, linked by NS 3000/XL, the left eight bits (0:8) are the remote environment of the connection. The right eight bits (8:8) are the LDEV of the device on the remote computer where the file label resides. ■ If the DS device for the RFA or the LDEV is 0, then a zero is returned. Note: If a zero is returned, use item 50.
7	U16	Hardware device address: Returns 2048. Maintained to provide backward compatibility with MPE V/E-based systems.
8	I16	File code: Returns the file code of a disk file (refer to `FFILEINFO` for file codes).
9	I32	Current logical record pointer: Returns the current logical record pointer setting. This value is the displacement in logical records from record number 0 in the file and identifies the record that would be accessed next by `FREAD` or `FWRITE`.
10	I32	EOF: Returns the pointer setting of the last logical record currently in the file (equivalent to EOF). If the file does not reside on disk, the value is zero. For message files, when a call is made to `FCONTROL` with *itemnum*=46, the number of records returned includes open, close, and data records.
11	I32	File limit: Returns a number representing the last logical record that can exist in the file (equivalent to the file limit). If the file does not reside on disk, the value is zero.
12	I32	Log count: Returns the logical records passed to and from the program during the current file access.
13	I32	Physical count: Returns the number of buffered physical I/O operations performed since the last `FOPEN/HPFOPEN` call (records).

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemmum	Item Type	Item Description
14	I16	Block size: Returns the file block size: ■ If the file is binary, the value is positive and the size is in halfwords. ■ If the file is ASCII, the value is negative and the size is in bytes. Maintained for compatibility with MPE V/E-based systems only. CM block size limits are used when `FGETINFO` returns block size information on all file types (STD, KSAM, RIO, CIR, MSG). If the block size of the specified file exceeds the limits, zero is returned. Note: If a zero is returned, use item 68.
15	I16	Extent size: Returns the extent size; for compatibility with MPE V/E-based systems only. Note: If a zero is returned, use item 69. If extent size is specified or the maximum number of extents is specified at file creation, the size and number of extents are determined by the operating system and the *item* values are not actual values; they are calculated using system defaults.
16	U16	Maximum number of extents: If the extent size or maximum number of extents is specified as zero at file creation, then the size and number of extents are determined by the system. In that case, these item values are calculated using system defaults defaults and do not reflect actual values.
17	I16	User labels: Returns the number of user labels defined for the file during creation. If the file is not a disk file, this number is zero. When an old file is opened for overwrite output, the value is not reset and the old user label is not destroyed.
18	CA	Creator: Returns the name of the file creator (at least 8 bytes). If the file does not reside on disk, blanks are returned. An unqualified form of the file owner's name is returned as the value of this *itemnum*. The file owner is not neccessarily the file's creator. File ownership may be changed using (see engineer). A symbolic zero (ASCII 48 in decimal) is returned as the file owner for root directories, accounts, and MPE groups created prior to the POSIX release. If the file is not located in the account in which the file owner is a member, a blank file owner name is returned. Item number 85 should be used to obtain the full file owner name instead of item 18.
19	I32	Label address: Returns a zero. For compatibility with MPE V/E-based systems only.

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
20	I16	Blocking factor
21	I16	Physical block size; indicates halfwords
22	I16	Data block size; indicates halfwords
23	I16	Offset to data in blocks; indicates halfwords
24	I16	Offset of active record table for RIO files; indicates halfwords
25	I16	Size of active record table within the block; indicates halfwords
26	CA	Volume ID (tape label)
27	CA	Volume set ID (tape label)
28	U16	Expiration date (julian format)
29	I16	File sequence number
30	I16	Reel number
31	I16	Sequence type
32	U16	Creation date (julian format)
33	I16	Label type
34	I16	Current number of writers
35	I16	Current number of readers
36	U16	File allocation date, when the file was last restored (`CALENDAR` format)
37	I32	File allocation time, when the file was last restored (`CLOCK` format)
38	U16	Spoolfile device file number: Bits (1:15) = Device file number Bit (0:1) = 1 Output spoolfile Bit (0:1) = 0 Input spoolfile If the spoolfile device number is larger than 32767, *itemnum* 38 returns 0 (zero). Use *itemnum* 78 instead for spoolfile numbers larger than 32767.
40	I32	Disk device status: Returns a zero. For compatibility with MPE V/E-based systems only.

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemmum	Item Type	Item Description
41	I16	Device type
42	I16	Device subtype: Always returns an 8. (Indicates a 7933 or 7935 disk drive)
43	CA	Environment file name (>=36 bytes)
44	I16	Number of disk extents currently allocated to the file
45	CA	File name from labeled tape header 1 record (>= 17 bytes)
46	I16	Tape density
47	I16	DRT number: Always returns an 8.
48	I16	Device unit number: Always returns a 0.
49	U16	Equivalent to a software interrupt PLABEL for message files
50	U16	Real device number of the file
51	I16	Remote environment number Note: If using NS 3000/XL RFA (remote file access), specify `DSDEVICE` *ldev#* when you are using a DS (point-to-point or X.25) link.
52	I32	Last modification time (`CLOCK` format) Zero is returned as the modification time for root directories, accounts, and MPE groups created prior to the POSIX release.
53	U16	Last modification date (`CALENDAR` format) Zero is returned as the modification time for root directories, accounts, and MPE groups created prior to the POSIX release.
54	U16	File creation date (`CALENDAR` format) Zero is returned as the modification time for root directories, accounts, and MPE groups created prior to the POSIX release.
55	U16	Last access date (`CALENDAR` format) Zero is returned as the modification time for root directories, accounts, and MPE groups created prior to the POSIX release.
56	I32	Number of data blocks in a variable length file
57	I16	Number of user labels written to the file
58	I16	Number of accessors having output access (write) for a particular file
59	I16	Number of accessors having input access (read/update) for a particular file
60	I16	Terminal type: 0 File's associated device not a terminal 1 Standard hardwire or multipoint terminal 2 Terminal connected through phone-modem 3 DS pseudo-terminal 4 X.25 Packed Switching Network PAD (packet assembler/disassembler) terminal 5 NS virtual terminal

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
61	CA	NS 3000/XL remote environment ID name Note: If using NS 3000/XL RFA (remote file access), specify `DSDEVICE` *ldev#* when using a DS (point-to-point or X.25) link. A buffer must be provided for the node name (or *envid*) with the required space of 52 bytes; otherwise, data corruption may occur on variables following *itemnum*=61 or an `FSERR 73, BOUNDS VIOLATION` may be returned.
62	CA	File lockword (8 bytes):
63	CA	Unique file identifier (UFID) (20 bytes):
64	@64	Virtual address of the file: Applicable for standard disk files only. (Requesting *itemnum*s 64, 74, or 75 for any other file type, RIO, MSG, CIR, causes an error and returns CCL (`1`).)
65		Reserved for the operating system.
66	@32	Virtual address of global unique file descriptor (GUFD):
67	U32	(NM) Record size (indicates bytes)
68	U32	Block size (indicates bytes)
69	U32	Extent size (indicates bytes)
74	@64	Virtual address of file label: Applicable for standard disk files only. (Requesting *itemnum*s 64, 74, or 75 for any other file type (RIO, MSG, CIR) causes an error and returns CCL (`1`).)
75	CA	Hardware path: Applicable for standard disk files only. (Requesting *itemnum*s 64, 74, or 75 for any other file type (RIO, MSG, CIR) causes an error and returns CCL (`1`).)
76	CA	Volume restriction (34 bytes): The last two characters indicate the type: 0 File placed on the specified volume at creation 1 File can be placed on any volume containing the specified class at creation 2 File can be placed on any volume within the specified volume set at creation (Default)
77	U32	Transaction management log set ID If *itemnum* 77 = 0 (zero), the file is not attached to the XM (Transaction Management) log.
78	U32	Spoolfile device file number: Bits (1:31) = Device file number Bit (0:1) = 1 Output spoolfile Bit (0:1) = 0 Input spoolfile

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemmum	Item Type	Item Description
79	I16	File's pending disposition 0 = No change, the disposition is the same as before the file was opened 1 = Permanent 2 = Temporary (tape files rewound) 3 = Temporary (same as 2 except tape files not rewound) 4 = Released (purged) 5 = Temporary (but the file was previously a permanent file)
80		This *itemnum* returns a null-terminated POSIX-syntax system absolute pathname for the file or directory referenced by *filenum*. On input the first four bytes of this buffer are interpreted as a 32-bit unsigned integer specifying the maximum buffer size in bytes. This maximum buffer size does not include the four bytes used to represent this size. On output the first four bytes of the buffer represent the pathname length excluding the null-terminator as an unsigned integer. The pathname is returned in the bytes following the pathname length. Bytes beyond the null-terminator should be considered undefined. If the maximum buffer length is incorrect on input, variables allocated near the buffer may be overwritten or a bounds violation may occur. A zero pathname length is returned for unnamed new files and when an error occurs. Zero is the mininum buffer length on input for this *itemnum*. (**add drawing of Format of the buffer on input**)
81		32-bit unsigned integer by reference. The current number of hard links to the file.
82		32-bit signed integer by reference. Time of last file access in clock format. The bit assignments are: Bits 0 — 7 hours Bits 8 — 15 minutes Bits 16 — 23 seconds Bits 24 — 31 tenths of seconds
83		32-bit signed integer by reference. Time of last file status change. (Clock format - See item 82 for a description of the format).
84		16-bit unsigned integer by reference. Date of last file status change in calendar format. The bit assignements are: Bits 0 - 7 — Year of the century Bits 8 - 15 — Day of the year
85		32-byte character array by reference. File Owner: The full file owner name. Unused characters are blank filled. A symbolic zero (ASCII 48 in decimal) is returned as the file owner for root directories, accounts, and MPE groups created prior to the POSIX release.

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
86		32-bit signed integer by reference. File owner identifier: The file owner identifier (UID). Zero is returned as the file owner ID for root directories, accounts, and MPE groups created prior to the POSIX release.
87		32-byte character array by reference. File group: The file group name. Unused characters are blank filled. A symbolic zero (ASCII 48 in decimal) is returned as the file group for root directories whose GID's have not been assigned.
88		32-bit signed integer by reference. File group identifier: The file group identifier (GID). Zero is returned as the file group ID for root directories whose GID's have not been assigned.
89		32-bit unsigned integer by reference. File type: The following valid file types may be returned: 0 Ordinary File 1 KSAM/3000 2 RIO 3 KSAM XL 4 CIR 5 Native Mode Spool File 6 MSG 7-8 Not Applicable 9 Directory 10-11 Not Applicable 12 Pipe 13 FIFO 14 Symbolic link 15 Device link

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemmum	Item Type	Item Description
90		32-bit unsigned integer by reference. Record type: The following valid record types may be returned: 0 Fixed 1 Variable 2 Undefined 3 Spool block 4 Root directory 5 Not applicable 6 Account directory 7 Group directory 8 Not applicable 9 Byte stream 10 Hierarchical directory
91		64-bit signed integer by reference. The current file size in bytes. The value returned represents the current position of the End-of-File (EOF) and may not reflect the number of bytes actually occupied by the file on disk if the file is sparsely allocated.
92		32-bit signed integer by reference. KSAM XL file version: This item returns a value indicating the version of a KSAM XL file. A value of 1 indicates an original type KSAM XL file, and a value of 2 indicates the next generation KSAM XL file. A value of zero is returned if the file is not a KSAM XL file.
93		32-bit unsigned integer by reference. NM Plabel: This item returns a 32-bit NM Plabel of a message file interrupt handler. Interrupts may be enabled on message files by calling the `FCONTROL` intrinsic with item 48 and the Plabel address.

Table 9-2. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
94		32-bit signed integer by reference. MPE/iX device type: This item returns the following values for the following types of devices: 0 Disk device 1 Tape device 2 Terminal device 3 Printer device 4 Remote device 5 Ports device 6 Reserved 7 Streams device 8 Sockets device
95		32-bit signed integer by reference. Close-on-Exec: This item returns a value indication whether or not this *filenum* is closed if one the POSIX.1 `exec()` family of functions if called. A value of 1 means that the file is closed on an `exec()` call, while a value of 0 indicates the file will survive across `exec()` calls.
96		32-bit signed integer by reference. POSIX Append mode: This item returns a value indicating whether or not this *filenum* has the POSIX.1 append mode flag set. When the append mode flag is set on files that support this feature, all writes occur at the end of the file, although reads may occur anywhere in the file. A value of 1 indicates that the POSIX.1 append mode is on, while a value of 0 indicates the append mode is off. The only time that the POSIX.1 append mode is valid is when a file has been oepned for byte stream access (`HPFOPEN` option 77 with a value of 2).
97		32-bit signed integer by reference. POSIX non-block mode: This item returns a value indicating whether or not this *filenum* has the POSIX.1 non-block flag set. When the non-block flag is set, on files that support this feature, reads, writes, and opens can be affected in a file dependent manner. In general, operations that would otherwise have impeded the caller results in immediate return when this flag is set. A value of 1 indicates the non-block flag is set, while a value of zero indicates the flag is not set. The only time the non-block flag is valid is for pipes and FIFO's.

Table 9-3. FFILEINFO File Codes

Integer	Mnemonic	Description
0		Default (unreserved)
1024	USL	User subprogram library
1025	BASD	Basic data
1026	BASP	Basic program
1027	BASFP	Basic fast program
1028	RL	Compatibility mode relocatable library
1029	PROG	Compatibility mode program file
1030	NMPRG	Native mode program file
1031	SL	Segmented library
1032	NMSL	Native mode executable library
1033	NMRL	Native mode relocatable library
1035	VFORM	VPLUS forms file
1036	VFAST	VPLUS fast forms file
1037	VREF	VPLUS reformat file
1040	XLSAV	Cross loader ASCII file (SAVE)
1041	XLBIN	Cross loader relocated binary file
1042	XLDSP	Cross loader ASCII file (DISPLAY)
1050	EDITQ	Edit quick file
1051	EDTCQ	Edit KEEPQ file (COBOL)
1052	EDTCT	Edit TEXT file (COBOL)
1054	TDPDT	TDP diary file
1055	TDPQM	TDP proof marked QMARKED
1056	TDPP	TDP proof marked non-COBOL file
1057	TDPCP	TDP proof marked COBOL file
1058	TDPQ	TDP work file
1059	TDPXQ	TDP work file (COBOL)
1060	RJEPN	RJE punch file
1070	QPROC	QUERY procedure file
1080	KSAMK	KSAM key file
1083	GRAPH	GRAPH specification file
1084	SD	Self-describing file

Table 9-3. FFILEINFO File Codes (continued)

Integer	Mnemonic	Description
1090	LOG	User logging log file
1100	WDOC	Hewlett-Packard WORD document
1101	WDICT	Hewlett-Packard WORD hyphenation dictionary
1102	WCONF	Hewlett-Packard WORD configuration file
1103	W2601	Hewlett-Packard WORD attended printer environment
1110	PCELL	IFS 3000/XL character cell file
1111	PFORM	IFS 3000/XL form file
1112	PENV	IFS 3000/XL environment file
1113	PCCMP	IFS 3000/XL compiled character cell file
1114	RASTR	Graphics image in RASTR format
1130	OPTLF	OPT/3000 log file
1131	TEPES	TEPE/3000 script file
1132	TEPEL	TEPE/3000 log file
1133	SAMPL	APS/3000 log file
1139	MPEDL	MPEDCP/DRP log file
1140	TSR	Hewlett-Packard Toolset root file
1141	TSD	Hewlett-Packard Toolset data file
1145	DRAW	Drawing file for Hewlett-Packard DRAW
1146	FIG	Figure file for Hewlett-Packard DRAW
1147	FONT	Reserved
1148	COLOR	Reserved
1149	D48	Reserved
1152	SLATE	Compressed SLATE file
1153	SLATW	Expanded SLATE work file
1156	DSTOR	RAPID/3000 DICTDBU utility store file
1157	TCODE	Code file for TRANSACT/XL compiler
1158	RCODE	Code file for Report/3000 compiler
1159	ICODE	Code file for Inform/3000 compiler
1166	MDIST	Hewlett-Packard Desk distribution list
1167	MTEXT	Hewlett-Packard Desk text

Table 9-3. FFILEINFO File Codes (continued)

Integer	Mnemonic	Description
1168	MARPA	ARPA messages file
1169	MARPD	ARPA distribution list
1170	MCMND	Hewlett-Packard Desk abbreviated commands file
1171	MFRTM	Hewlett-Packard Desk diary free time list
1172	None	Reserved
1173	MEFT	Hewlett-Packard Desk external file transfer messages file
1174	MCRPT	Hewlett-Packard Desk encrypted item
1175	MSERL	Hewlett-Packard Desk serialized (composite) item
1176	VCSF	Reserved
1177	TTYPE	Terminal type file
1178	TVFC	Terminal vertical format control file
1192	NCONF	Network configuration file
1193	NTRAC	Network trace file
1194	NLOG	Network log file
1195	MIDAS	Reserved
1211	ANODE	Reserved
1212	INODE	Reserved
1213	INVRT	Reserved
1214	EXCEP	Reserved
1215	TAXON	Reserved
1216	QUERF	Reserved
1217	DOCDR	Reserved
1226	VC	VC file
1227	DIF	DIF file
1228	LANGD	Language definition file
1229	CHARD	Character set definition file
1230	MGCAT	Formatted application file

Table 9-3. FFILEINFO File Codes (continued)

Integer	Mnemonic	Description
1236	BMAP	Base map specification file
1242	BDATA	BASIC data file
1243	BFORM	BASIC field order file for VPLUS
1244	BSAVE	BASIC saved program file
1245	BCNFG	Configuration file for default option BASIC program
1258	PFSTA	Pathflow static file
1259	PFDYN	Pathflow dynamic file
1270	RFDCA	Revisable form DCA data stream
1271	FFDCA	Final form DCA data stream
1272	DIU	Document interchange unit file
1273	PDOC	Hewlett-Packard WORD/150 document
1401	CWPTX	Reserved
1421	MAP	Hewlett-Packard MAP/3000 map specification file
1422	GAL	Reserved
1425	TTX	Reserved
1461	NMOBJ	Native mode object file
1462	PASLB	Pascal/XL source library

FOPEN FOPTIONS

Bits	(0:1)	(1:1)	(2:3)		(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
Field	Reserved	Record Format Extension	File Type		Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default File Designator	ASCII BINARY	Domain
Meaning		0= don't use extended record format 1= use extended record format – only valid for (8:2 = 01 → byte stream)	00 00 01 01 10 10 11	0=STD 1=KSAM 0=RIO 1=KSAMXL 0=CIR 1=SPOOLFILE 0=MSG	0=Allow :FILE 1=No :FILE	0=Non–Labeled Tape 1=Labeled Tape	0= NOCCTL 1=CCTL	00=Fixed 01=Variable or Byte Stream 10=Undefined 11=Spoolfile	000=FILENAME 001=$STDLIST 010=$NEWPASS 011=$OLDPASS 100=$STDIN 101=$STDINX 110=$NULL	0= BINARY 1= ASCII	00=New File 01=Old Permanent File 10=Old Temporary File 11=Old Permanent or Temporary File

LG200154_002c

NOTE: Double lines indicate octal digit boundaries

Figure 9-1. Foption Bit Summary

FOPEN AOPTIONS

Bits	(0:3)	(3:1)	(4:1)	(5:2)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)	
Field	Reserved	File Copy Access	NOWAIT I/O	Multi–Access	Inhibit Buffering	Exclusive Access	Dynamic Locking	Multi–Record Access	Access Type	
Meaning		0=Access Native Mode 1= Access as Standard Sequential File	1=NOWAIT 0= Non–NOWAIT	00= Non Multi–Access 01= Only Intra–Job Multi–Access 10= Inter–Job Multi–Access Allowed	0= BUF 1=NOBUF	00= Default 01= Exclusive 10= Semi–exclusive Access Read 11= Share	0= No FLOCK Allowed 1= FLOCK Allowed	0= No Multi–Record 1= Multi–Record	0 000=Read only 0 001=Write only 0 010=Write (Save) only 0 011=Append only 0 100=Read/Write 0 101=Update 0 110=Execute 1 001=Directory Read Access	

LG200154_003a

NOTE: Double lines indicate octal digit boundaries

Figure 9-2. Aoption Bit Summary

Condition Codes

CCE (2) Request granted.

CCG (0) Not returned.

CCL (1) Request denied. Access or calling sequence error.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FFINDBYKEY

Positions the record pointer at the beginning of the first record matching the key value comparison in a KSAM file.

Syntax

```
                I16V   CA   I16V  I16V I16V
FFINDBYKEY(filenum,value,location,length,relop);
```

Parameters

filenum **16-bit integer by value (required)**

Identifies the file number of the file to be positioned.

value **character array (required)**

Contains a value that determines which record is read. This value is compared to the data contained in *location* in relation to the operator specified in *relop*.

location **16-bit integer by value (required)**

Specifies the relative byte location in the record of the key being used. Bytes are numbered starting with 1. If *location*=0, the primary key is used.

length **16-bit integer by value (required)**

Specifies the length of the key in bytes. If *length*=0, then the entire key is used. If *length* is less than the full key length (generic key), then only the length specified is used in the comparison with *relop*. The *length* parameter must be equal to or less than the full length of the key when the file was created. For numeric display keys or packed decimal keys, the full key length must be used.

relop **16-bit signed integer by value (required)**

Specifies the relational operator for the comparison of the key value of the file to the value specified in *value*. The record where the file is positioned has this relation to key value:

Value	Meaning
0	Equal
1	Greater than
2	Greater than or equal to

When *relop* is set to 1 or 2, the search is for an approximate key.

Operation Notes

Split stack calls are permitted.

The `FFINDBYKEY` intrinsic does not read the advance flag. It positions both the logical record pointer and the physical pointer to the appropriate record. When the function is complete, it sets the advance flag to FALSE.

To locate and read a single record, use the `FREADBYKEY` intrinsic.

Condition Codes

CCE	Request granted.
CCG	Request denied. The requested position was beyond the logical end-of-file or beginning-of-file.
CCL	Request denied. An error occurred: an I/O error occurred, the *relop* parameter could not be satisfied, a *length* less than the full length was specified for a key with numeric display or packed decimal format, or a key was not found when *relop*=0.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FFINDN

Positions the logical record pointer to the relative record number according to the key sequence in a KSAM file.

Syntax

```
          I16V     DV      I16V
FFINDN(filenum,number,location);
```

Parameters

filenum **16-bit signed integer (required)**

Passes the file number of the file to be positioned.

number **double by value (required)**

Specifies a record number relative to the first logical record in the file. Record numbers start with zero or one depending on the record numbering scheme specified at file creation. The lowest numbered record applies to the record with the lowest value in the specified key field. A negative record number positions the file pointer to the record with the smallest key value.

location **16-bit signed integer by value (required)**

Passes the relative byte location in the record of the key to be used. The first byte of the record is considered 1. If *location*=0, the primary key is used.

Operation Notes

Split stack calls are permitted.

This intrinsic does not read the advance flag. It sets both the logical record pointer and the physical pointer to the appropriate record. When its function is complete, it sets the advance flag to FALSE.

When the relative record number is specified, be sure not to confuse this number with the physical record number (the number of the record as it is stored in the file). The relative record number is based on the value of a specified key, not its location in a file.

If `FFINDN` is used to position the pointer before calling another procedure that reads or updates the file in a shared environment, `FLOCK` must be called before calling `FFINDN`. After performing the read or update operation, unlock the file. If the file is locked after calling `FFINDN`, another user can change the pointer position without your program being aware of it.

Condition Codes

CCE	Request granted.
CCG	Request denied. The requested position was beyond the logical end-of-file.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FGETINFO

Returns access and status information about a file.

Note

FGETINFO is provided for compatibility with MPE V/E-based systems only. It is recommended that FFILEINFO be used to access data.

Syntax

```
                I16V        CA        U16    U16
FGETINFO(filenum,formaldesig,foption,aoption
                I16     I16     U16     U16    I16
          lrecsize,devtype,ldevnum,hdaddr,filecode,
                I32   I32 I32       I32     I32     I16
          lrecptr,eof,filelimit,logcount,physcount,blksize,
                U16       I16       I16        CA     I32
          extsize,numextent,userlabels,creatorid,labaddr);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file for which information is requested.

formaldesig **character array (optional)**

Returns the actual designator of the file being referenced, in the following format:

filename.groupname.accountname

The *formaldesig* array must be at least 28 bytes in length. When the actual designator is returned, unused bytes in the array are filled with blanks on the right. A nameless file returns an empty string.

foption **16-bit unsigned integer by reference (optional)**

Returns seven different file characteristics by setting corresponding bit groupings. The file characteristics are those specified for *foptions* in the FOPEN intrinsic.

aoption **16-bit unsigned integer by reference (optional)**

Returns up to seven different access options represented by bit groupings as described for the *aoptions* parameter of FOPEN.

lrecsize **16-bit signed integer by reference (optional)**

Returns the logical record size associated with the file:

- If the file was created as a binary file, this value is positive and expresses the size in halfwords.

- If the file was created as an ASCII file, this value is negative and expresses the size in bytes.

devtype **16-bit signed integer by reference (optional)**

Returns the type and subtype of the device being used for a KSAM, RIO, circular, or message file, or devices such as a tape drive, printer, or terminal where bit (0:8) indicate device subtype, and bit (8:8) indicate device type. For standard disk files, bit (8:8)=00000011 and bit (0:8)=00001000 (indicate a 7933/35 disk drive).

ldevnum **16-bit unsigned integer by reference (optional)**

Returns the logical device number (ldev) associated with the device where the file label resides:

- If the file is a disk file, *ldevnum* is the location of the file label. (File data may reside on the same device as the file label.)
- If the file is spooled, *ldevnum* is a virtual device number that does not correspond to the system configuration I/O device list.
- If the file is located on a remote computer, linked by a DS point-to-point or X.25 link, the left eight bit (0:8) are the logical device number of the distributed system (DS) device.
- If the remote computer is linked by NS 3000/XL, the left eight bit (0:8) are the remote environment of the connection. The right eight bit (8:8) are the ldev of the device on the remote computer where the file label resides.
- If the DS device for the RFA or the LDEV is 0, then *ldevnum* returns a 0.

hdaddr **16-bit unsigned integer by reference (optional)**

Returns 2048. Maintained to provide backward compatibility with MPE V/E-based systems.

filecode **16-bit signed integer by reference (optional)**

Returns the file code of a disk file.

lrecptr **32-bit signed integer by reference (optional)**

Returns the current physical record pointer setting. Remember that physical record numbers can begin with zero or one, depending on how the file was built.

eof **32-bit signed integer by reference (optional)**

Returns the pointer setting of the last logical record currently in the file (equivalent to the number of logical records currently in the file). If the file does not reside on disk, this value is zero. For interprocess communication (IPC), when a call to `FCONTROL` with *itemnum*=46 is in effect, the number of records returned in *eof* includes open, close, and data records.

filelimit **32-bit signed integer by reference (optional)**

Returns a number representing the last logical record that could exist in the file (the physical limits of the file). If the file does not reside on disk, this value is zero.

logcount **32-bit signed integer by reference (optional)**

Returns the total number of logical records passed to and from the program during the current file access.

physcount **32-bit signed integer by reference (optional)**

Returns the total number of physical I/O operations performed within the process, against the file, since the last `FOPEN/HPFOPEN` call.

blksize **16-bit signed integer by reference (optional)**

Returns the file block size:

- If the file is binary, the value is positive and the size is in halfwords.
- If the file is ASCII, the value is negative and the size is in bytes.

extsize **16-bit unsigned integer by reference (optional)**

Maintained to provide backward compatibility with MPE V/E-based systems.

numextent **16-bit signed integer by reference (optional)**

Maintained to provide backward compatibility with MPE V/E-based systems.

userlabels **16-bit signed integer by reference (optional)**

Returns the number of user labels defined for the file during creation. If the file is not a disk file, this number is zero. When an old file is opened for overwrite output, the value of *userlabels* is not reset, and old user labels are not destroyed.

creatorid **character array (optional)**

Returns the name of the file creator (8-character array). If the file is not a disk file, blanks are returned.

labaddr **32-bit signed integer by reference (optional)**

Returns a zero. Maintained for backward compatibility with MPE V/E-based systems.

Operation Notes

Returns access and status information about a file located on any device. The file must be opened by the calling process at the time of the `FGETINFO` call.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FGETKEYINFO

Requests access and status information about a KSAM file.

Syntax

```
                I16V    BA      BA
FGETKEYINFO(filenum,param,control)
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file about which information is requested.

param **byte array (required)**

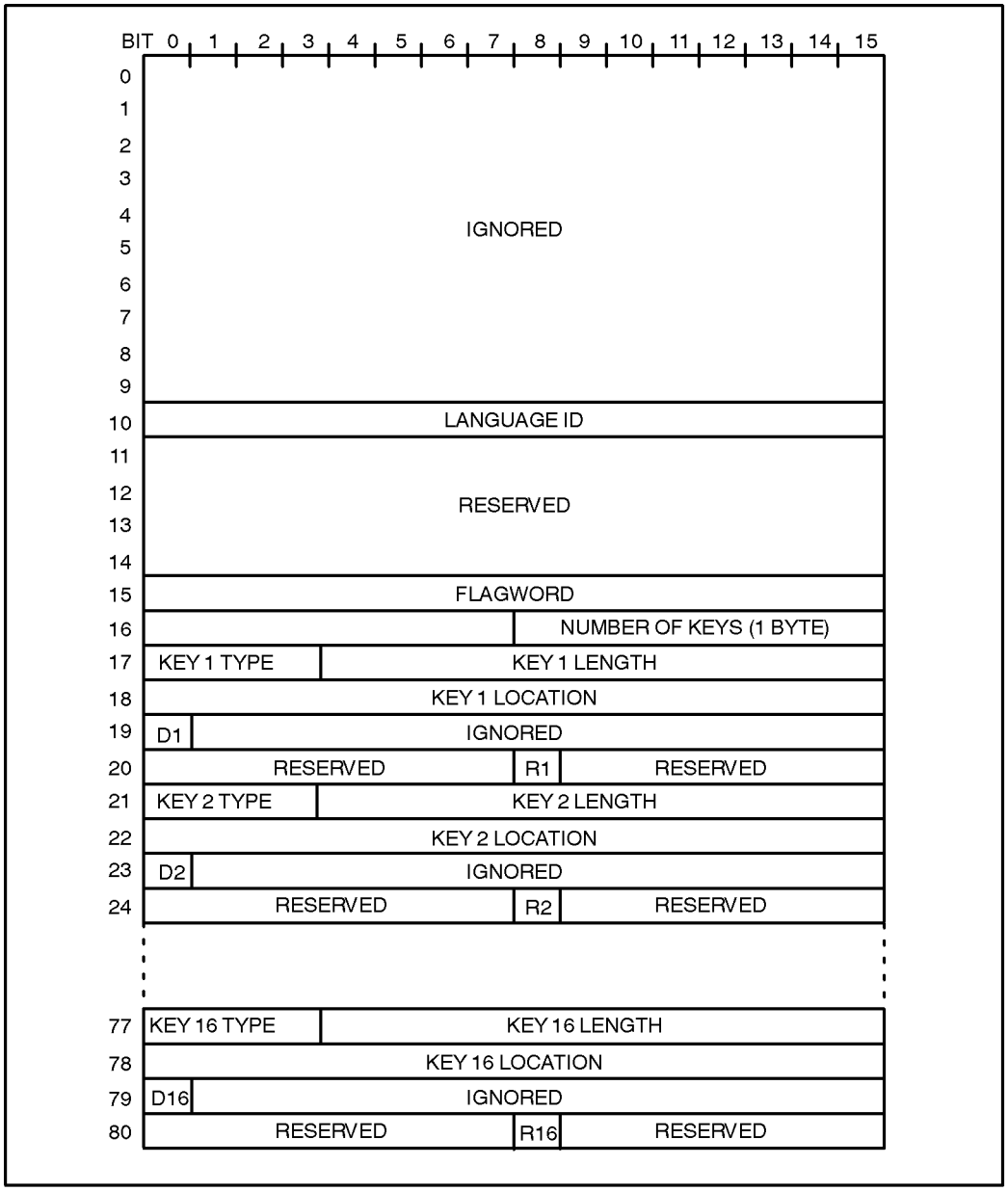
Returns information describing the key information for a KSAM file. The length is 162 bytes.

control **byte array (required)**

Passes 256 bytes of control information about the key file.

Operation Notes

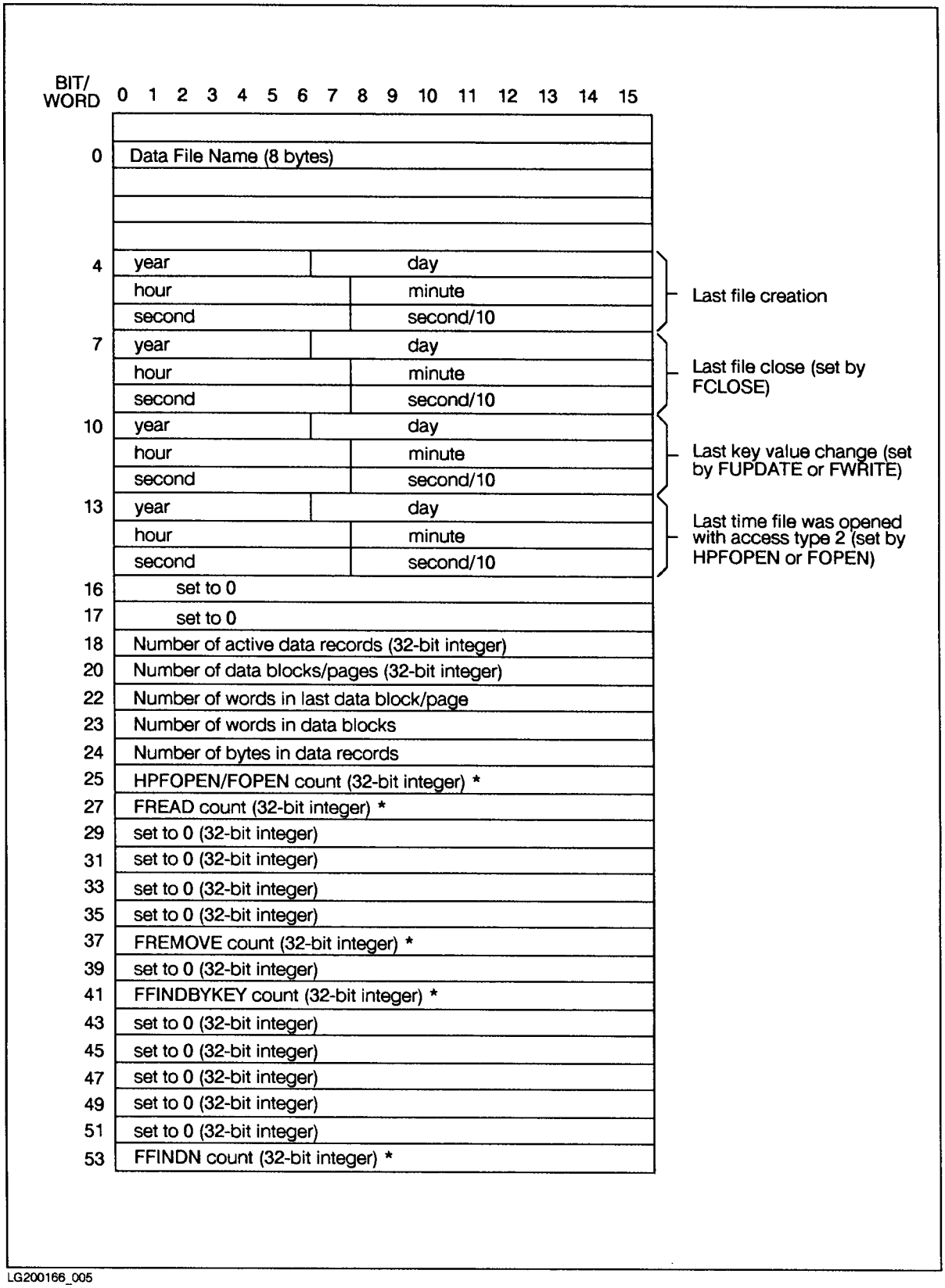
The `FGETKEYINFO` parameter returns an array equivalent to the array for the `HPFOPEN` and `FOPEN` intrinsics. (Refer to Figure 9-3.) Its length must be 162 bytes.



LG200166_004

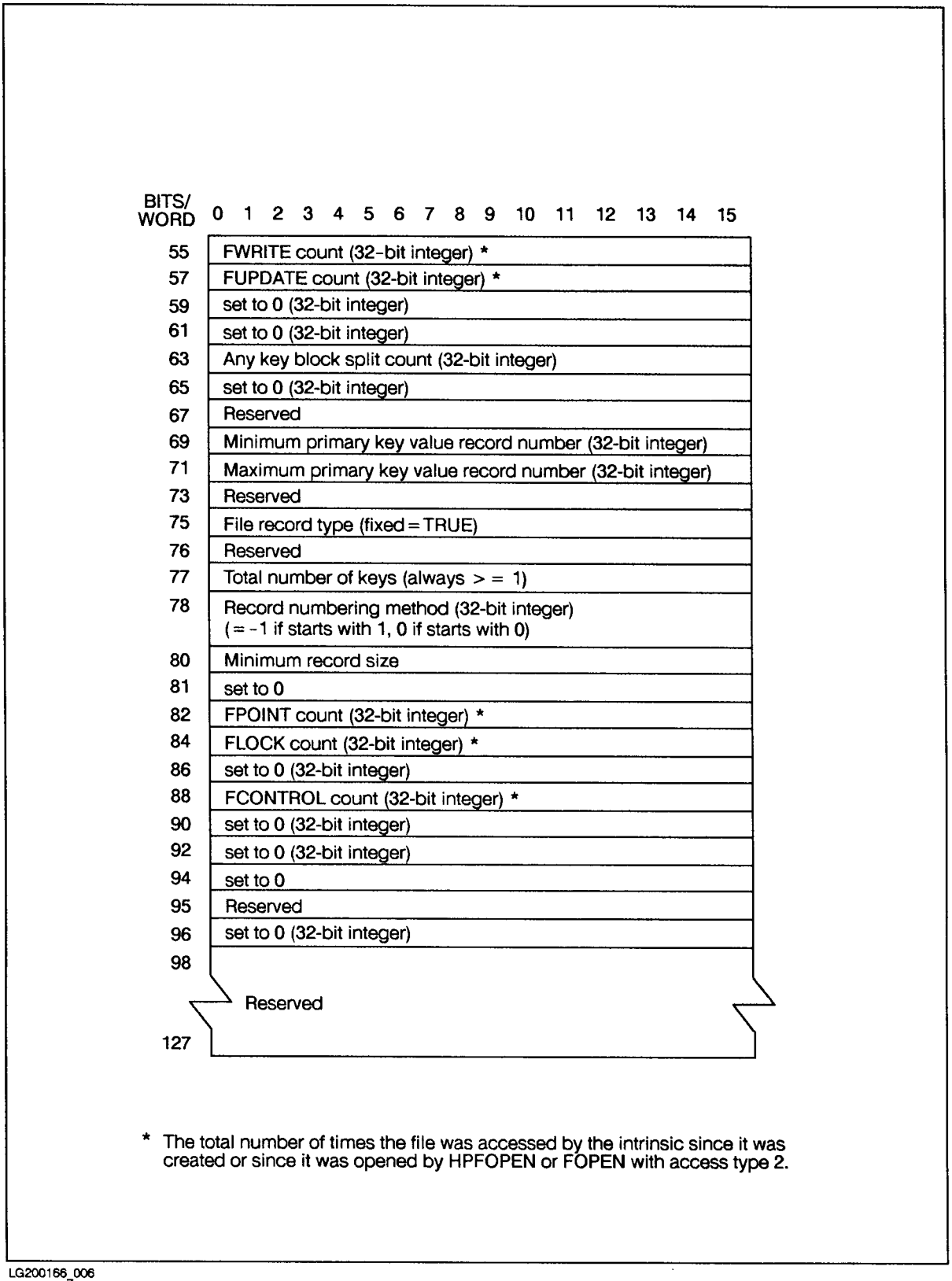
Figure 9-3. FGETKEYINFO Parameter Format

The *control* parameter provides dynamic information about the use of the file from the time it was created. It counts the number of times the file was referred to by intrinsics, and the date and time it was created, closed, updated, or written to. Its format is shown in Figure 9-4.



LG200166_005

Figure 9-4. FGETKEYINFO Control Parameter Format



LG200166_006

FGETKEYINFO Control Parameter Format (continued)

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. An error occurred; insufficient space was declared for *param* or *control*, an illegal file number was specified, or the DB register is not set to the user stack.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FLABELINFO

Returns information from the file label of a disk file.

Syntax

```
                    CA        I16V      I16
FLABELINFO(formaldesig,mode,fserrorcode,
          I16A    REC    I16A
   itemnum,item,itemerror);
```

Parameters

formaldesig **character array (required)**

Passes the name of the file using either MPE syntax (the default) or HFS syntax. The file name must be terminated by a nonalphanumeric character other than a period (.), a slash (/), a hyphen (-), and an underscore (_).

If MPE syntax, the file name can include password, group, and account specifications. The file name can backreference a file equation and optionally be preceded by an asterisk.

If HFS syntax, the file name must start with either a dot (.) or a slash (/). For files located in HFS directories, traverse directory entries (TD) access is required to all directories specified in *formaldesig*. If there is no TD access, `FLABELINFO` fails and a file system error code (398) is returned in the *fserrorcode* parameter.

If the file can be named using both MPE syntax and HFS syntax (for example, `FILEA.MYGROUP.MYACCT` and `/MYACCT/MYGROUP/FILEA`), the file can be either permanent or temporary. If a temporary and a permanent file have the same name, `FLABELINFO` returns information about the temporary file only.

mode **16-bit signed integer by value (required)**

Passes an option specifying the valid backreferencing to file equations for the file. Valid values are:

Value	Meaning
0	Use file equation (if one exists)
1	Must use file equation (error if one does not exist)
2	Ignore existing file equations

Bits Value/Meaning

0:11 Reserved for future use.

12:1 Symbolic Link Traversal

- 0 To traverse through symbolic links, if they exist.
- 1 Do not traversing through symbolic links, if they exist.

13:2 Caller Privilege Level Allows the caller to pretend to be less privileged. The privilege level is passed in this field.

15:2 File Equations

- 0 Use file equations if they exist.
- 1 A file equation must be used.
- 2 Do not use a file equation.

fserrorcode **16-bit signed integer by reference (required)**

Returns a value indicating whether an error or warning occurred when `FLABELINFO` attempted to return requested information:

- A value of zero indicates that no errors were encountered.
- A positive value is a file system error code and indicates that an error was encountered and no information was returned in *item*.
- A -1 indicates that an item error or warning has occurred. Check the *itemerror* parameter to determine which item(s) has an error/warning and what it is.

itemnum **16-bit signed integer array (required)**

Specifies which *item* value is to be returned. (Refer to Table 9-4.)

To indicate the end of the list, place a zero in the element following the last *itemnum*.

item **record (required)**

Returns the value of the item specified in the corresponding *itemnum*. (Refer to Table 9-4.)

*Itemnum/item*s are paired such that the *n*th field of the *item* record corresponds to the *n*th element of the *itemnum* array.

itemerror **16-bit signed integer array (required)**

Returns an error number corresponding to the items specified in the *itemnum* array. The *itemnum/item* and *itemerror* parameters are paired such that the *nth* element of the *itemerror* array corresponds to the *nth* element of the *itemnum* array.

If a value in the *itemerror* array is negative, a warning exists for the corresponding item. If the value is positive, an error was detected for the corresponding item. The absolute value of each value is a file system error number.

Table 9-4. FLABELINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	CA	File name (8 bytes): The file name component for the file referenced in *formaldesig* is returned as the value. If the file name is not expressible using MPE-only semantics, a file system error code (391) is returned in the associated *itemerror*.
2	CA	Group name (8 bytes): The group name component for the file referenced in *formaldesig* is returned as the value. If the group name is not expressible using MPE-only semantics, a file system error code (391) is returned in the associated *itemerror*.
3	CA	Account name (8 bytes): The account name component for the file referenced in *formaldesig* is returned as the value. If the account name is not expressible using MPE-only semantics, a file system error code (391) is returned in the associated *itemerror*.
4	CA	File creator name (8 bytes): An unqualified form of the file owner's name is returned as the value. The file owner is not necessarily the file's creator. A symbolic zero (ASCII 48 in decimal) is returned as the file owner for root directories, MPE accounts, and MPE groups created prior to release 4.5. If the file is not located in the account where the file owner is a member, a blank file owner name is returned. Use *itemnum*=43 to obtain the full file owner name.
5	U32	Security matrix for access: Returns the file's security matrix. This value does not indicate the actual security enforced for a file, since group and account security masks can also restrict access. This field is ignored if an ACD is active on a file.
6	U16	File creation date: The date in `CALENDAR` intrinsic format. Either creator (C) or manager (AM if file is within account, otherwise SM) access required. Zero is returned as the creation date for root directories, MPE accounts, and MPE groups created prior to release 4.5.
7	U16	Last access date: The date in `CALENDAR` intrinsic format. May not be up-to-date when the file is open. Zero is returned as the last access date for root directories, MPE accounts, and MPE groups created prior to release 4.5.
8	U16	Last modification date: The date in `CALENDAR` intrinsic format. May not be up-to-date when the file is open. Zero is returned as the modification date for root directories, MPE accounts, and MPE groups created prior to release 4.5.
9	I16	File code of disk file
10	U16	Number of user labels written: May not be up-to-date when the file is open.
11	U16	Number of user labels available: May not be up-to-date when the file is open.
12	I32	Total number of logical records possible in the file: Equivalent to the file limit measured in logical records.

Table 9-4. FLABELINFO Itemnum/Item Values (continued)

Itemmum	Mnemonic	Item Description
13	U16	File options: The record format extension bit is returned as the foption (1:1) bit. Byte stream record format is represented as a record format extension of one with a variable record format (foption (8:2) bits equal to 01). Directories, symbolic links, device links, pipes and FIFO's cannot be represented by *foption*. If the object referenced by `filenum` is is an object, MPE error `399` is returned in the associated *itemerror*. Refer to the *foption* figure.
14	I16	Record size: Maintained for compatibility with MPE V/E-based systems. (If a zero is returned, use *itemnum* 30 instead.)
15	I16	Block size: Maintained for compatibility with MPE V/E-based systems. (If a zero is returned, use *itemnum* 31 instead.)
16	I16	Maximum number of extents: Maintained for compatibility with MPE V/E-based systems. (If a zero is returned, use *itemnum* 32 instead.)
17	I16	Last extent size: Indicates sectors. May not be up-to-date when the file is open.
18	I16	Extent size: Indicates sectors. (If a zero is returned, use *itemnum* 32 instead.)
19	U32	Number of logical records in file: Equivalent to EOF. May not be up-to-date when the file is open.
20	U32	File allocation time: The time when file was last restored (in `CLOCK` intrinsic format). Zero is returned as the file allocation time for root directories, MPE accounts, and MPE groups created prior to release 4.5.
21	U16	File allocation date: The date when the file was last restored (in `CALENDAR` intrinsic format). Zero is returned as the file allocation date for root directories, MPE accounts, and MPE groups created prior to release 4.5.
22	I32	Number of open/close records: MSG files only. May not be up-to-date when the file is open.
23	CA	Device name (8 bytes)
24	U32	Last modification time: The time when the file was last modified (in `CALENDAR` intrinsic format). May not be up-to-date when the file is open.
25	CA	First user label (user label 0) (256 bytes): May not be up-to-date when the file is open. Manager (AM if file is within account, otherwise SM) or read/write (R/W) access required.
27	REC	Unique file identifier (UFID) (20 bytes)
28	U32	Total number of bytes allowed in file: Equivalent to the file limit measured in bytes. May not be up-to-date when the file is open.
29	U32	Start of file offset: Indicates the byte offset where user data starts.

Table 9-4. FLABELINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
30	U32	Record size (indicates bytes
31	U32	Block size (indicates bytes)
32	U32	Extent size (indicates bytes)
33	CA	File lockword (8 bytes): Returned if you are the file creator, account manager, or system manager.
34	CA	Volume restriction (34 bytes): The last two characters indicate the type of restriction, as follows: 0 File is placed on the specified volume at creation 1 File can be placed on any volume containing the specified class at creation 2 File can be placed on any volume within the specified volume set at creation (Default)
35	CA	Volume set names (32 bytes): No restrictions.
36	CA	Transaction management log set id (4 bytes) No restrictions.
37	U16	Logical device number
38	REC	Terminated HFS-syntax system absolute pathname: Upon input, the first four bytes are interpreted as a 32-bit unsigned integer specifying the maximum available buffer size in bytes. This maximum available buffer size does not include the four bytes used to represent this size. Upon output, the first four bytes represent the pathname length excluding the null terminator as a 32-bit unsigned integer. The pathname is returned in bytes following the pathname length. Bytes beyond the pathname terminator are undefined. If the maximum available buffer size is incorrect upon input, variables allocated near the buffer can be overwritten or a bounds violation could occur. A zero pathname length is returned for unnamed new files and when an error occurs. Zero is the minimum buffer length upon input for this *itemnum*.
39	U32	The current number of hard links to the file

40	I32	Time of last file access (clock format): The bit assignments are:
		`bits 0-7 = hours`
		`bits 8-15 = minutes`
		`bits 16-23 = seconds`
		`bits 24-31 = tenths of seconds`

41	I32	Time of last file status change (clock format): The bit assignments are:
		`bits 0-7 = hours`
		`bits 8-15 = minutes`
		`bits 16-23 = seconds`
		`bits 24-31 = tenths of seconds`

42	U16	Date of the last file status change (calendar format): The bit assignments are:
		`bits 0-7 = year of century`
		`bits 8-15 = day of the year`

Table 9-4. FLABELINFO Itemnum/Item Values (continued)

Itemmum	Mnemonic	Item Description
43	CA	File owner (32 bytes): The full file owner name. Unused characters are filled with blanks. A symbolic zero (ASCII 48 in decimal) is returned as the file owner for root directories, accounts, and MPE groups created prior to release 4.5.
44	I32	File owner identifier: The file owner identifier (UID). Zero is returned as the file owner ID for root directories, MPE accounts, and MPE groups created prior to release 4.5.
45	CA	File group (32 bytes): The file group name. Unused characters are filled with blanks. A symbolic zero (ASCII 48 in decimal) is returned as the group for root directories where GIDs have not been explicitly assigned.
46	I32	File group identifier: The file group identifier (GID). Zero is returned as the group ID for root directories where GIDs have not been explicitly assigned.
47	U32	File type: Following are valid file types that can be returned: 0 = Ordinary file 1 = KSAM/3000 2 = RIO 3 = KSAM XL 4 = CIR 5 = Native Mode Spool File 6 = MSG 7 = N/A 8 = N/A 9 = Directory 10-11= N/A 12 = Pipe 13 = FIFO 14 = Symbolic Link 15 = Device Link
48	U32	Record type: Following are valid record types that can be returned: 0 = fixed 1 = variable 2 = undefined 3 = spool block 4 = root directory 5 = N/A 6 = account directory 7 = group directory 8 = N/A 9 = byte stream 10 = hierarchical directory
49	I64	Current file size (in bytes): The value returned represents the current position of the end-of-file (EOF) and may not reflect the number of bytes actually occupied by the file on disk if the file is sparsely allocated.

Table 9-4. FLABELINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
50	I32	KSAM XL File Version: This item returns a value indicating the version number of a KSAM XL file. A value of 1 indicates an original type KSAM XL file. A value of 2 indicates the next generation KSAM XL file. A value of zero is returned if the file is not a KSAM XL file.
51	I32	KSAM XL Parameters: This item returns file information about KSAM XL.
52	I32	MPE/iX Device Type: This item returns the following values for the following types of devices: 0=Disk device 1=Tape device 2=Terminal device 3=Printer device 4=Remote device 5=Ports device 6=Reserved 7=Streams device 8=Sockets device
53	I32	Secure/Release: This item returns a value indicating whether the file is currently secured or released. A value of 1 indicates that the file is secured. A value of zero indicates that the file is released.

FOPEN FOPTIONS

Bits	(0:1)	(1:1)	(2:3)		(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
Field	Reserved	Record Format Extension	File Type		Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default File Designator	ASCII BINARY	Domain
Meaning		0= don't use extended record format 1= use extended record format – only valid for (8:2 = 01 → byte stream)	00 00 01 01 10 10 11	0=STD 1=KSAM 0=RIO 1=KSAMXL 0=CIR 1=SPOOLFILE 0=MSG	0=Allow :FILE 1=No :FILE	0=Non–Labeled Tape 1=Labeled Tape	0= NOCCTL 1=CCTL	00=Fixed 01=Variable or Byte Stream 10=Undefined 11=Spoolfile	000=FILENAME 001=$STDLIST 010=$NEWPASS 011=$OLDPASS 100=$STDIN 101=$STDINX 110=$NULL	0= BINARY 1= ASCII	00=New File 01=Old Permanent File 10=Old Temporary File 11=Old Permanent or Temporary File

LG200154_002c

NOTE: Double lines indicate octal digit boundaries

Figure 9-5. Foption Bit Summary

Condition Codes

CCE (2)	Request granted.
CCG (0)	Not returned.
CCL (1)	Request denied. An error occurred. Refer to the *fserrorcode* and *itemerror* parameters for more information.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FLOCK

Dynamically locks a file. A call to FLOCK is required before any attempt is made to read or modify a file with shared access.

Note

The file system does not guarantee exclusive access, even when FLOCK and FUNLOCK are used, unless all programs that access the file cooperate by using locking. A program that opens the file with dynamic locking enabled will still be allowed to modify the file, even if it never calls FLOCK.

Syntax

```
        I16V  U16V
FLOCK(filenum,lockflag);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file whose global resource identification number (RIN) is to be locked.

lockflag **16-bit unsigned integer by value (required)**

Specify either conditional or unconditional locking by setting bit (15:1) as follows:

Value	Meaning
0	Locking takes place only if the file's global RIN is not currently locked. If the RIN is locked, control returns immediately to the calling process, with condition code CCG.
1	Locking takes place unconditionally. If the file cannot be locked immediately, the calling process suspends until the file can be locked.

Condition Codes

The following condition codes are possible when *lockflag* bit (15:1)=1:

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. This file was not opened with the dynamic locking *aoption* bit (10:1) specified in the FOPEN/HPFOPEN intrinsic.

The following condition codes are possible when *lockflag* bit (15:1)=0:

CCE	Request granted.
CCG	Request denied because the file was locked by another process.

CCL	Request denied. This file was not opened with the dynamic locking *aoption* bit (10:1) specified in the `FOPEN/HPFOPEN` intrinsic.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FOPEN

Opens a file.

Syntax

```
  I16                CA      U16V  U16V  I16V     CA
filenum:=FOPEN(formaldesig,foption,aoption,recsize,device,
                     CA     I16V
               formmsg,userlabels
                 I32V     I16V      I16V    I16V
               filesize,numextent,initialloc,filecode);
```

Functional Return

filenum — **16-bit signed integer (assigned functional return)**

Returns a unique file number identifying the opened file.

Parameters

formaldesig — **character array (optional)**

Passes a formal file designator, following file naming conventions. The file name must begin with a letter and contain alphanumeric characters, slashes, or periods. Terminate the string by placing a delimiter in the array element following the last valid character. The delimiter can be any nonalphanumeric character except a slash (/), period (.), colon (:), or exclamation point (!).

If the file name is the name of a user-defined file, it can begin with an asterisk (*). If the file name is the name of a system-defined file, it can begin with a dollar sign ($). The remote location of a device can be specified as *filename:envid*. The file, lockword, group, and account names are each limited to eight characters in length.

The formal file designator can contain command interpreter variables and expressions that are evaluated before the formal file designator is parsed and validated.

Default: A nameless file is assigned that can be read or written to, but not saved. (The domain option of a nameless file must specify a new file unless it is a device file.)

foption — **16-bit unsigned integer by value (optional)**

Specifies up to eight different file characteristics, as noted below, by setting corresponding bit groupings:

Note

For existing files, default conditions are specified in the file label.

Device characteristics may override some *foptions*.

Bits	Value/Meaning
14:2	Domain

Indicates which file domain is searched to locate a file. A nameless disk file must always be a new file. A device file (such as a tape or terminal) always resides in the system file domain (permanent file directory). Always specify a device file as old or permanent.

The following bit settings are valid:

00 The file is new. No search is necessary.

01 The file is a permanent file. The system file domain (permanent file directory) is searched.

10 The file is a temporary file. The job file domain (temporary file directory) is searched.

11 The file is an old (permanent or temporary) file. The job file domain (temporary file directory) is searched. If not found, the system file domain is searched.

Default: 00

13:1 ASCII/binary

Indicates which code, ASCII or binary, a new file is in when written to a device that supports both codes. This option is applicable only at file creation. type

The following bit settings are valid:

0 Binary file

1 ASCII file

Default: 0

10:3 Designator

The actual file designator is the same as the formal file designator (000). This is the default and only setting allowed for KSAM files.

8:2 Record format

Bit settings indicate internal record structure for a file. This option is applicable only at file creation.

KSAM XL supports fixed-length records only (00). The file contains logical records of uniform length.

7:1 Carriage control

No carriage-control directive is expected for KSAM files.

5:1 Disallow file equation option

Indicates whether or not to allow file equations. A leading * in a formal file designator can override the setting to disallow FILE. The following bit settings are valid:

0 Allow FILE equations to override programmatic or system-defined file specifications.

1 Disallow FILE equations from overriding programmatic or system-defined file specifications.

Default: 0

2:3 File type option

Indicates internal record structure used to access records in a file. KSAM XL files are identified by a setting of 011.

0:2 Reserved for MPE/iX

aoption **16-bit unsigned integer by value (optional)**

Specifies up to eight different file access options, as noted below, by setting corresponding bit groupings:

Bits **Value/Meaning**

12:4 Access type

Indicates the type of access intended for the file. This option restricts usage of file system intrinsics.

The following bit settings are valid:

0000 Allows read access only, provided that the file's security provisions specify read access. FWRITE, FUPDATE, and FREMOVE intrinsic calls cannot reference this file. The end-of-file (EOF) is not changed.

0001 Allows write access only, provided that the file's security provisions allow write access. Any data written in the file prior to the current FOPEN request is deleted. FFINDBYKEY, FFINDN, FPOINT, FREAD, FREADBYKEY, FREADC, FREADDIR, FREMOVE, FSPACE, and FUPDATE intrinsic calls cannot reference this file. The EOF is set to 0.

0010 Allows write-save access only, if the file's security provisions allow write access. Previous data in the file is not deleted. FFINDBYKEY, FFINDN, FPOINT, FREAD, FREADBYKEY, FREADC, FREADDIR, FREMOVE, FSPACE, and FUPDATE intrinsic calls cannot reference this file. The EOF is not changed. Therefore, data is overwritten if a call to FWRITE is made. The system changes this value to append for message files.

0011 Allows append access only, if the file's security provisions allow either append or write access. FFINDBYKEY, FFINDN, FPOINT, FREAD, FREADBYKEY, FREADC, FREADDIR, FREMOVE, FSPACE, and FUPDATE intrinsic calls cannot reference this file. For disk files, the EOF is updated after each FWRITE call. Therefore, data cannot be overwritten.

0100 Allows read/write (I/O) access only, provided that the file's security provisions allows both read and write access. If both read and write access are not allowed, the access type specified in the security provisions (either read or write) is allowed. Any file intrinsic except FUPDATE and FREMOVE can be called for this file. The EOF is not changed. This option is not valid for message files.

0101 Allows update access only, if the file's security provisions allows both read and write access. If both read

and write access are not allowed, the access type specified in the security provisions (either read or write) is allowed. All file intrinsics can be called for this file. The EOF is not changed. This option is not valid for message files.

0110 Allows execute access only, if the file's security provisions allow execute access. This access allows read/write access to any loaded file. The program must be running in PM to specify execute access. This option is not valid for message files.

0111 Allows execute/read access only, if the file's security provisions allow execute access. This access allows only read access to any loaded file. The program must be running in PM to specify execute/read access. This access is changed to execute (only) access for KSAM, CIR, and RIO files. This option is not valid for message files.

Default: 0000

10:1 Dynamic locking

Enables/disables file locking for the file. When this option is specified, the `FLOCK` and `FUNLOCK` intrinsics can be used to dynamically permit or restrict concurrent access to a disk file by other processes at specified times.

The following bit settings are valid:

0 Disallow dynamic locking/unlocking.

1 Allow dynamic locking/unlocking.

Default: 0

If several accessors are sharing the file, they must all specify, or not specify, this option. For example, if a file is opened with the dynamic locking option enabled, and a subsequent accessor tries to open the file with dynamic locking disabled, the subsequent attempt to open fails.

Note

The file system does not guarantee exclusive access, even when FLOCK and FUNLOCK are used, unless all programs that access the file cooperate by using locking. A program that opens the file with dynamic locking enabled will still be allowd to modify the file, even if it never calls FLOCK.

8:2 Exclusive option

Indicates continuous exclusive access to this file, from open to close. Use this option when performing a critical operation (for example, updating the file).

The following bit settings are valid:

00 If access type option (*aoption* bit (12:4)) specifies read only access, then read-share access takes effect. Otherwise, exclusive access takes effect. Regardless of which access option was selected, FFILEINFO reports zero.

01 Exclusive access. After the file is opened, any additional HPFOPEN/FOPEN requests for this file are prohibited until this process issues the FCLOSE request or terminates. If any process is already accessing this file when an HPFOPEN/FOPEN call is issued with exclusive access specified, an error status is returned. If another HPFOPEN/FOPEN call is issued for this file while exclusive access is in effect, an error code is returned to the process that issued the call. Request exclusive access only if the lock access mode is allowed by the security provisions for the file.

10 Read-share access (semi-exclusive access). After the file is opened, concurrent write access to this file through another HPFOPEN/FOPEN request is prohibited, whether issued by this process or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the read/write or update access type option (*aoption* bit (12:4)) obtains read access. However, other types of read access are allowed. If a process already has write access to the file when this call is issued, an error code is returned

to the calling process. If another HPFOPEN/FOPEN call that violates the read only restriction is issued while read-share access is in effect, that call fails and an error code is returned to the calling process. Request read-share access only if the lock access mode is allowed by the security provisions for the file.

11 Share access. After the file is opened, concurrent access to this file by any process is permitted, in any access mode, subject to other security provisions in effect.

Default: 00

5:2 Multiaccess mode option KSAM XL supports no multiaccess (00).

Default: 00

4:1 NOWAIT I/O option KSAM XL does not support NOWAIT I/O (0).

Default: 0

3:1 Copy mode option Determines whether a file should be treated as a standard sequential file (copy by logical record) or physical block (copy to another file).

KSAM XL does not allow the copy mode option (0).

Default: 0

0:3 Reserved for MPE/iX.

recsize **16-bit signed integer by value (optional)**

Passes the size, in halfwords or bytes, of the logical records in the file. Positive values are halfwords, negative values are bytes. The valid range is dependent on storage and record formats:

- For fixed-length and undefined-length ASCII files, the valid range is 1 to 32,767 bytes.
- For variable-length ASCII files and fixed-length, variable-length, and undefined-length binary files, the range is 1 to 32,766 bytes (1 to 16,383 halfwords). All odd values specified are rounded up to the next even value (the next halfword boundary).

Default: Device dependent.

device **character array (optional)**

Passes a string of ASCII characters terminating with any nonalphanumeric character except a slash (/) or period (.), designating the *device* where the file is to reside. For a KSAM file, the device must be a random access device such as a disk.

Default: `DISC`

ksamparam **character array (optional)**

Contains a description of the KSAM XL parameters including the primary key and up to 15 alternate keys. If a new file is being created, this parameter must be specified. If this is an existing file, check flag word field to see if the default values are acceptable. In the flag word field you can set bit 13 to sequential write. For COBOL, set flag 9. If this is not an existing file, specify this field explicitly. (Refer to Figure 9-6 for parameter format.)

Language ID Number

This three-digit code identifies the native language to be used for the file. To display a list of native languages that are available on your system, enter `RUN NLUTIL.PUB.SYS`.

If the file already exists, this field is ignored.

Flag word

The flag word contains a halfword defining the file characteristics.

Bits	Value/Meaning
15:1	Reserved, do not use. Always set to 0.
14:1	Enter 1 if record numbering is to start with 1. Enter 0 if record numbering is to start with 0.
13:1	Enter 1 if only sequential writing by primary key is allowed. Enter 0 if random writing by primary key is allowed.
12:1	Enter 1 if deleted record space can be reused. Enter 0 if deleted record space cannot be used.
11:1	Enter 1 if a language type is specified. Enter 0 if a language type is not specified.
10:1	Enter 1 if the primary key cannot be changed with the `FUPDATE` intrinsic for files that are opened for sequential processing. Enter 0 if the primary key can be changed

with the FUPDATE intrinsic for files that are opened for sequential processing. This enables KSAM processing of COBOL information according to COBOL standards.

9:1 Enter 1 if the file is programmatically accessed by the COBOL programming language. Enter 0 if the file is not programmatically accessed by the COBOL programming language. This enables KSAM to process COBOL information according to COBOL standards.

8:1 Enter 1 if selecting optimal block size.

0:9 Enter 0. These bits are reserved and must contain zeros.

Number of Keys

In bits 8:8, enter a number between 1 and 16 specifying the number of keys to be defined for this file.

Key Definitions

Each key in the file requires a 4-halfword word definition. The first definition is always the primary key. Up to 15 alternate keys are allowed for any KSAM XL file. The key definitions contain the key type, key length, key location, duplicate key flag, and random insert flag:

Key Type

Bits 0:4 specify the type of key:

Value	Meaning
0001	Byte key (1 to 255 bytes)
0010	Short integer key (255 bytes)
0011	Integer key (255 bytes)
0100	Real number key (255 bytes)
0101	Long real number key (255 bytes)
0110	Numeric display key (1 to 28 bytes)
0111	Packed decimal key (1 to 14 bytes)
1000	Signed packed decimal key (2 to 14 bytes)
1001	IEEE floating-point decimal key (4, 8, or 16 bytes)

Key Length

Bits 4:12 specify the key length. Enter the length of the key in bytes. A maximum of 255 bytes is allowed, but the length is dependent on the type of key data specified.

Key Location

Enter the relative location in bytes of the key field in the record. Note that the first byte of the record is considered 1.

Duplicate Key Flag

Bits 0:1 specify the duplicate key flag. Enter 1 if duplicate key values are allowed for this key. Enter 0 if duplicate key values are not allowed for this key.

Random Insert Flag

Bits 8:1 specify the random insert flag. This field specifies the method of inserting duplicate key values. To use this feature, the previous duplicate key flag must be set to 1. Bits 0:8 and 9:7 are reserved and always set to 0.

Enter 1 if duplicate key values are to be inserted randomly in the duplicate key chain.

Enter 0 if duplicate key values are to be inserted at the end of the duplicate key chain.

userlabels **16-bit signed integer by value (optional)**

Passes the number, in the range 0 to 254, of user-label records to be created for the file. Applicable to new disk files only.

Default: 0

filesize **32-bit signed integer by value (optional)**

Passes the maximum file capacity.

KSAM XL requires extra space for its index area. The actual space needed is computed by the KSAM XL type manager, based on the file size specified by the user. If the space required to build a file of the user-specified size exceeds 2 gigabytes, `FOPEN` returns an error.

numextent **16-bit signed integer by value (optional)**

Passes a value in the range 1 to 32 that determines the number of extents for the file. If a value of 1 and an *initialloc* value of 1 is specified, the file is created as one contiguous extent of disk space. If a value >1 is specified, a variable number of extents (with varying extent sizes) are allocated on a need basis. Applicable only at file creation.

Default: >=1 extents

initialloc **16-bit signed integer by value (optional)**

Passes an integer value in the range 1 to 32 that determines the number of extents to be allocated to the file initially. Applicable only at file creation.

Default: 0

filecode **16-bit signed integer by value (optional)**

Passes a value that can be used as a file code to identify the type of file. This code is recorded in the file label and is accessible through the `FFILEINFO` intrinsic. Applicable only at file creation (except when opening an old file that has a negative file code).

If the program is running in user mode, specify a file code in the range 0 to 32,767 to indicate the file type being created; programs running in user mode can access files with nonnegative file codes. If the program is running in privileged mode, specify a file code in the range -32,768 to 32,767; programs running in privileged mode can access files with a file code in the range -32,768 to 32,767. If an old file with a negative file code is opened, the file code specified must match the file code in the file label.

Default: 0

Table 9-5. FOPEN/HPFOPEN Parameter Equivalents

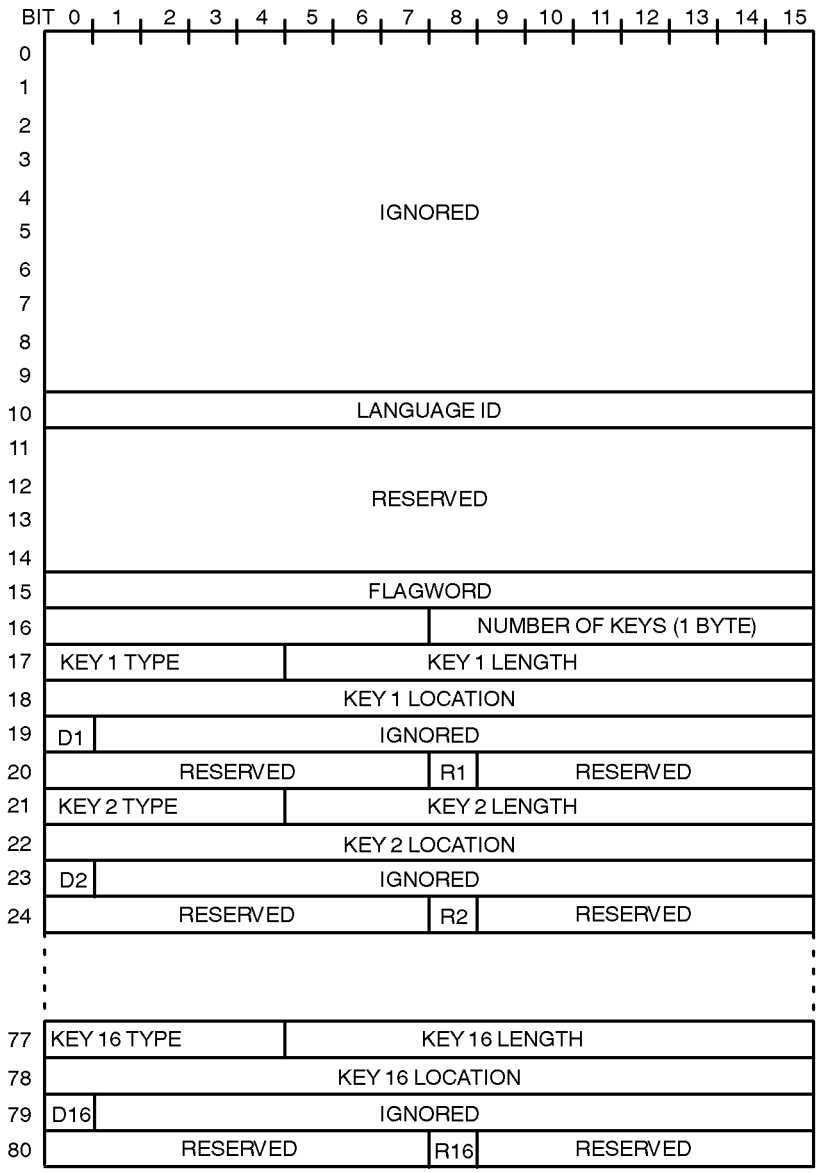
FOPEN Parameter	HPFOPEN Itemnum,Item
filenum (functional return)	*filenum* (parameter)
formaldesig	*2,formaldesig*
foption: Bits (14:2) Domain Bit (13:1) ASCII/binary Bits (10:3) File designator Bits (8:2) Record format Bit (7:1) Carriage-control Bit (6:1) Labeled tape Bit (5:1) Disallow file equation Bits (2:3) File type	 *3, domain* *53, ASCII/binary* *5, file designator* *6, record format* *7, carriage-control* *8, labeled tape* *9, disallow file equation* *10, file type*
aoption: Bits (12:4) Access type Bit (11:1) Multirecord Bit (10:1) Dynamic locking Bits (8:2) Exclusive Bit (7:1) Inhibit buffering Bits (5:2) Multiaccess mode Bit (4:1) Nowait I/O Bit (3:1) File copy	 *11, access type* *15, multirecord* *12, dynamic locking* *13, exclusive* *46, inhibit buffering* *14, multiaccess mode* *16, nowait I/O* *17, file copy*
recsize	*19, record size*
device	*20, device name* *22, volume class* *23, volume name* *24, density* *25, printer environment* *26, remote environment* *42, device class* *48, reverse VT*
formmsg	*8, labeled tape label* *28, spooled message* *30, labeled tape type* *31, labeled tape expiration* *32, labeled tape sequence* *54, KSAM parms*
userlabels	*33, user labels*
blockfactor	*40, block factor*

Table 9-5.
FOPEN/HPFOPEN Parameter Equivalents (continued)

FOPEN Parameter	HPFOPEN Itemnum,Item
numbuffers: Bits (11:5) Numbuffers Bits (4:7) Spooler copies Bits (0:4) Output priority	 *44, numbuffers* *34, spooler copies* *27, output priority*
filesize	*35, filesize*
numextent	*47, numextent*
initialloc	*36, initial allocation*
filecode	*37, filecode*

Operation Notes

Figure 9-6 shows the format of the KSAM parameter.



LG200166_004

Figure 9-6. FOPEN KSAM XL Parameter Format

A file can be referenced by its formal file designator. When executed, a unique file number is returned to the process. This file number, rather than the formal file designator, is used in subsequent calls to this file.

Condition Codes

CCE	Request granted. The file is open.
CCG	Not returned.
CCL	Request denied. For example, another process already has exclusive or semi-exclusive access for this file, the privilege level of this file is not user (3), or an initial allocation of disk space cannot be made due to lack of disk space. If the file is not opened successfully, the file number value returned by `FOPEN` is 0. Call the `FCHECK` intrinsic for more details.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FPOINT

Sets the logical and physical record pointers to the specified record.

Syntax

```
          I16V     I32V
FPOINT(filenum,lrecnum);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file where the pointer is to be set.

lrecnum **32-bit signed integer by value (required)**

Passes the relative physical record number where the physical record pointer is to be positioned. Record numbering starts with zero or one, depending on how the file was created.

Operation Notes

This intrinsic does not read the advance flag. It positions both the logical record pointer and the physical pointer to the appropriate record. When its function is complete, it sets the advance flag to FALSE.

Condition Codes

CCE — Request granted.

CCG — Request denied. The physical record pointer position is unchanged. Positioning was requested at a point beyond the file limit.

CCL — Request denied. The physical record pointer position is unchanged because of one of the following:

- Invalid *filenum* parameter.
- The *lrecnum* parameter specified a record marked for deletion.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREAD

Reads a logical record in key sequence from a file to the buffer.

Syntax

```
I16          I16V  UDS  I16V
lgth:=FREAD(filenum,buffer,length);
```

Functional Return

lgth — **16-bit signed integer (assigned functional return)**

Returns the length of the data transferred to *buffer*:

- If a negative value is passed in the *length* parameter, the *lgth* is a positive value indicating the number of bytes transferred.
- If a positive value is passed in the *length* parameter, the *lgth* is a positive value indicating the number of halfwords transferred.
- If a value of 0 is passed in the *length* parameter, the position is identified, but the data is not returned.

Parameters

filenum — **16-bit signed integer by value (required)**

Passes the file number of the file to be read.

buffer — **user-defined structure (required)**

Returns the record that was read. This structure must be large enough to hold all of the information to be transferred.

length — **16-bit signed integer by value (required)**

Passes the length of the data to be transferred to *buffer*. If this value is positive, it signifies the length in halfwords. If negative, it signifies the length in bytes. If zero, no transfer occurs.

If *length* is larger than the size of the logical record, transfer is limited to the length of the logical record. If less than the size of the logical record, the transfer is limited to the length specified.

Operation Notes

This intrinsic reads the advance flag and advances to the next record if the flag is set to TRUE. It positions the logical record pointer and the physical pointer to the appropriate record. When its function is complete, it sets the advance flag to TRUE.

When the logical end-of-data is encountered, CCG is returned to the process.

Condition Codes

CCE	Request granted. The information was read.
CCG	Request denied. The logical end-of-data was encountered during reading.
CCL	Request denied. The information was not read because an error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREADBYKEY

Reads a logical record based on key value from a KSAM file to the target.

Syntax

```
I16                 I16V     LA   I16V   CA
lgth:=FREADBYKEY(filenum,buffer,length,value,
          I16V
       location);
```

Functional Return

lgth — **16-bit signed integer by value (assigned functional return)**

Returns the length of the information transferred.

- If *lgth* is positive, it is a halfword count.
- If *lgth* is negative, it is a byte count.
- If *lgth* is 0, the position is identified, but the data is not returned.

Parameters

filenum — **16-bit signed integer by value (required)**

Passes the file number of the file to be read.

buffer — **logical array (required)**

Returns the transferred record. It must be large enough to hold all the information to be read.

length — **16-bit signed integer by value (required)**

Passes the number of halfwords or bytes to be transferred. If *length* is positive, it is the length in halfwords. If negative, it is the length in bytes. If zero, no transfer occurs.

If *length* is less than the size of the record to be transferred, only the first *length* halfwords or bytes are transferred from the record. If the *length* is larger than the physical record size, only the physical record length is transferred.

value — **character array (required)**

Passes the key value determining the record to be read. The first record found with an identical key value specified by *location* is the record read.

location — **16-bit signed integer by value (required)**

Passes the relative byte location in the record of the key whose value determines which record is to be read. The first byte is numbered as 1. If 0 is specified, the primary key is used.

Operation Notes

This intrinsic does not read the advance flag. It positions the logical record pointer and the physical pointer to the appropriate record. When its function is complete, it sets the advance flag to FALSE.

Condition Codes

CCE	Request granted.
CCG	Request denied. The logical end-of-data or beginning-of-data was encountered during the read.
CCL	Request denied. An error occurred. Either an I/O error occurred or the key could not be located.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREADC

Reads a logical record in physical sequence from a KSAM file to the target.

Syntax

```
I16           I16V   LAI  16V
lgth:=FREADC(filenum,buffer,length);
```

Functional Return

lgth — **16-bit signed integer by value (assigned functional return)**

Returns the length of the information transferred.

- If *lgth* is positive, it is a halfword count.
- If *lgth* is negative, it is a byte count.
- If *lgth* is 0, the position is identified, but the data is not returned.

Parameters

filenum — **16-bit signed integer by value (required)**

Passes the file number of the file to be read in physical record sequence.

buffer — **logical array (required)**

Returns the transferred record. It must be large enough to hold all the information to be read.

length — **16-bit signed integer by value (required)**

Passes the number of halfwords or bytes to be transferred. If *length* is positive, it is the length in halfwords; if negative, it is the length in bytes. If, zero, no transfer occurs.

If *length* is less than the size of the record to be transferred, only the first *length* halfwords or bytes are transferred from the record. If the *length* is larger than the physical record size, only the physical record length is transferred.

Operation Notes

This intrinsic reads the advance flag and advances to the next record if the flag is set to TRUE. It positions only the physical record pointer to the appropriate record. Deleted records are skipped. When its function is completed, it sets the advance flag to TRUE.

Condition Codes

CCE	Request granted.
CCG	Request denied. The logical end-of-data was encountered during the read.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREADDIR

Reads a logical record located by its physical record number from a file to the buffer.

Syntax

```
                I16V  UDS   I16V  I32V
FREADDIR(filenum,buffer,length,lrecnum);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be read.

buffer **user-defined structure (required)**

Returns the record that was read. This structure should be large enough to hold all of the information to be transferred.

length **16-bit signed integer by value (required)**

Passes the number of halfwords or bytes to be transferred. If this value is positive, it signifies halfwords. A negative value indictates a transfer in bytes. If zero, no transfer occurs.

If *length* is less than the size of the logical record, only the first *length* halfwords or bytes are read from the record. If *length* is larger than the size of the logical record, the transfer is limited to the length of the logical record.

lrecnum **32-bit signed integer by value (required)**

Indicates the relative physical record number to which the physical pointer is positioned. Physical record numbering for fixed-length records starts with zero or one, as specified when the file was built.

Operation Notes

This intrinsic reads the advance flag. It sets only the physical pointer to the appropriate record. When its function is completed, it sets the advance flag to TRUE.

This intrinsic is different from the FREAD intrinsic. The FREAD intrinsic reads only the record already pointed to by the logical record pointer. FREADDIR inputs the specified logical record. If the record is inactive, the contents of the inactive record are transmitted and a CCE is returned. There is no indication of the block containing some inactive records. (FCHECK returns a nonzero error number to distinguish active and inactive records.)

Condition Codes

CCE	Request granted. The information was read.
CCG	Request denied. End-of-data was encountered.
CCL	Request denied. The information was not read; an error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREADLABEL

Reads a user-defined file label.

Syntax

```
                I16V   UDS   I16V  I16V
FREADLABEL(filenum,buffer,length,labelid);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file whose label is to be read.

buffer **user-defined structure (required)**

Returns the label that was read. This structure must be large enough to hold the number of halfwords specified by *length*.

length **16-bit signed integer by value (optional)**

Passes the number of halfwords to be transferred from the label. This field must not be greater than 128 halfwords.

Default: 128 halfwords

labelid **16-bit signed integer by value (optional)**

Passes the label number. (The first label is numbered zero.)

Default: Zero

Operation Notes

When a disk file is opened, user labels can be read from it, or written to it, in any order, at any time, regardless of access capabilities to the rest of the file. A disk file can have as many as 254 128-halfword user-defined labels.

Condition Codes

CCE	Request granted. The label was read.
CCG	Request denied. A label was referenced beyond the last label written on the file.
CCL	Request denied. The label was not read; an error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FREMOVE

Marks the current record in a KSAM file for deletion.

Syntax

```
          I16V
FREMOVE(filenum)
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file where the record is to be deleted.

Operation Notes

Split stack calls are permitted.

When executed, the first bit in the record header is set to 1.

This intrinsic does not read the advance flag. It sets the logical record pointer and the physical physical pointer to the appropriate record. When its function is completed, it sets the advance flag to FALSE. When a record is deleted, the pointers are positioned at the next sequential record of the specified key.

Condition Codes

CCE	Request granted.
CCG	Request denied. The logical end-of-data was encountered.
CCL	Request denied. An error was encountered, the record is not deleted.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FRENAME

Renames an open disk file (and its lockword, if applicable). The file being renamed must be either:

- A new file.
- An old file (permanent or temporary), opened for exclusive access with the *exclusive* option of the HPFOPEN/FOPEN intrinsics, and with security provisions allowing write access.

Syntax

```
            I16V     CA
FRENAME(filenum,formaldesig);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be renamed.

formaldesig **character array (required)**

Passes the new name of the file. The maximum number of characters allowed in the string is 36. The ASCII string contained in *formaldesig* must begin with a letter and can contain up to eight alphanumeric characters for each of the *filename, lockword, group*, and *account* fields. The string must end with a nonalphanumeric character, including a blank, but not a slash (/) or a period (.). The home volume set of *formaldesig* must be the same as the file being renamed. Volume sets cannot be spanned when renaming files. The format of *formaldesig* is:

filename/lockword.group.account

where:

filename Is the new file name for the file. (Required in *formaldesig*.)

lockword Is a lockword for the new file name. (Optional portion of *formaldesig*.) To keep or add a lockword to the file, the *lockword* must be entered in the ASCII string. If this part of *formaldesig* is not specified, the new file name has no lockword associated with it.

group Is the group where the file is to reside. (Optional portion of *formaldesig*.) If a group is not specified, the file resides in the group it was assigned before the FRENAME intrinsic call.

account Is the account name where the file is to reside. (Optional portion of *formaldesig*.) If renaming a new or temporary file that was created, specify any account that shares the same volume set as the file being renamed. A permanent file cannot be renamed across account boundaries. If other than the current account name is specified for a permanent file, the CCL (1) error condition is returned and the file retains its old name.

Operation Notes

The *formaldesig* parameter uses MPE-escaped semantics. If a file is referenced by *filenum*, you can renamed it within the hierarchical directory as long as the process invoking `FRENAME` has sufficient access and the restrictions are satified. `FRENAME` intrinsic fully qualifies the file owner name. Only file owners and users with appropriate privilege can manipulate a file's lockword.

If renaming a file, a process must have the following:

`TD` Traverse directory entry to access to all directories specified in *formaldesig*. If *formaldesig* is specified as `file.group.account`, the directories are the root directory, the account, and the MPE group.

`CD` Create directory entry to access to the new parent directory.

`DD` Delete directory entry to access to the old parent directory.

`SF` Save files capability.

The following restrictions apply to `FRENAME`:

- Directories cannot be renamed.
- Lockwords cannot be assigned to hierarchical directories.
- Files cannot be renamed across volume sets.
- Files with KSAM/3000, RIO, and CIR file types may only be assigned names in the MPE name space.

If a file without an ACD is renamed from an MPE group to a directory (although not within the same account), an ACD is automatically assigned to the file.

All errors will set the condition codes to CCL.

CM KSAM files cannot be renamed, but KSAM XL files can.

Condition Codes

CCE (2)	Request granted.
CCG (0)	Not returned.
CCL (1)	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FSPACE

Moves a record pointer forward or backward in a file.

Syntax

```
          I16V       I16V
FSPACE(filenum,displacement);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file on which spacing is to be done.

displacement **16-bit signed integer by value (required)**

Passes the number of logical records to be spaced over, relative to the current position of the logical record pointer.

A positive value signifies forward spacing, a negative value signifies backward spacing. The maximum positive value is 32,767. The maximum negative value is −32,768.

Operation Notes

The logical record pointer is repositioned in key sequence. The spacing is based on the primary key unless an alternate key has been specified in a prior call to `FFINDN`, `FFINDBYKEY`, or `FREADBYKEY`.

This intrinsic reads the advance flag and advances to the next record if the flag is set to TRUE. It sets the logical record pointer and the physical pointer to the appropriate record. When its function is completed, it sets the advance flag to FALSE.

Note that because this intrinsic reads the advance flag, spacing might be affected by a preceding call to an `FREAD` or `FREADC` intrinsic. `FREAD` and `FREADC` set the advance flag to TRUE. If the `FSPACE` intrinsic is then called, it advances one record before moving back or ahead the specified number of records.

Condition Codes

CCE	Request granted.
CCG	Request denied. A logical end-of-file indicator was encountered during spacing. The logical record pointer is at the beginning-of-file if displacement was negative or at the end-of-file if displacement was positive.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FUNLOCK

Dynamically unlocks a file.

Syntax

```
          I16V
FUNLOCK(filenum);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file whose global RIN is to be unlocked.

Condition Codes

CCE	Request granted.
CCG	Request denied. The file had not been locked by the calling process.
CCL	Request denied. The file was not opened with the dynamic locking *aoption* of the `FOPEN/HPFOPEN` intrinsic, or the *filenum* parameter is invalid.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FUPDATE

Updates the contents of a logical record in a file.

Syntax

```
            I16V    UDS    I16V
FUPDATE(filenum,buffer,length);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be updated.

buffer **user-defined structure (required)**

Passes the record to be written in the update.

length **16-bit signed integer by value (required)**

Passes the number of halfwords or bytes to be written to the file. A positive value is in halfwords; a negative value is in bytes.

If *length* is less than record size, the length is transferred in halfwords or bytes and remaining portions of the record will be padded with fill characters. If *length* equals zero, no transfer occurs and the record address is overwritten with default fill characters (blanks for ASCII files; null characters for binary files). If *length* is greater than record size, CCL is returned and no transfer occurs.

Operation Notes

This intrinsic does not read the advance flag. If the record's key data is unchanged, it does not position any pointers, but sets the advance flag to TRUE. If the record's key data changes, it positions the logical record pointer and the physical pointer to the appropriate record and sets the advance flag to FALSE. The act of updating the keys advances the pointers to the next record.

The record to be updated is the record pointed to by the logical data pointer. `FUPDATE` moves the specified information from the stack into this record. The file containing this record must be opened with the update *aoption* specified in the `FOPEN/HPFOPEN` call and the file cannot have variable-length records. If RIO access is used, the modified record is set to the `ACTIVE` state.

Condition Codes

CCE	Request granted.
CCG	Request denied. An end-of-file condition was encountered during updating.
CCL	Request denied. An error occurred. The *length* exceeds the size of the record, *length* does not include all the keys, or a disk I/O error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FWRITE

Writes a logical record from the buffer to a file.

Syntax

```
          I16V   UDS   I16V   U16V
FWRITE(filenum,buffer,length,controlcode);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be written on.

buffer **user-defined structure (required)**

Passes the record to be written.

length **16-bit signed integer by value (required)**

Passes the number of halfwords or bytes to be written to the record. If this value is positive, it signifies halfwords; if negative, bytes. Zero indicates that no transfer occurs.

If *length* is less than the record size, the remaining portion of the record is padded with the fill character that is specified during the file creation. The default for ASCII is blank. The default for binary is binary zero.

If *length* is larger than the logical record size, the `FWRITE` request is refused and CCL is returned.

controlcode **16-bit unsigned integer by value (required)**

This parameter must be specified to satisfy internal requirements, but it is ignored.

Operation Notes

This intrinsic does not read the advance flag. It positions the logical record pointer and the physical pointer to the appropriate record. When its function is completed, it sets the advance flag to FALSE.

When the `FWRITE` intrinsic is executed, the logical record pointer is set to the record immediately following the record just written. When an `FWRITE` call writes a record beyond the current logical end-of-file indicator, this indicator is advanced. If the physical bounds of the file are reached, CCG is returned.

Condition Codes

CCE	Request granted.
CCG	Request denied. The physical bounds of the file prevented further writing.
CCL	Request denied. An error occurred: an I/O error occurred;

- a duplicate key value occurred when duplicates are not allowed
- *length* does not include all keys
- sequential processing was specified in the flag word of the *ksamparam* in `FOPEN` and the primary key is not in ascending order.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

FWRITELABEL

Writes a user-defined file label.

Syntax

```
                I16V    UDS  I16V  I16V
FWRITELABEL(filenum,buffer,length,labelid);
```

Parameters

filenum **16-bit signed integer by value (required)**

Passes the file number of the file to be labeled.

buffer **user-defined structure (required)**

Passes the label to be written. If the file is a labeled magnetic tape file, this label must be 40 halfwords in length.

length **16-bit signed integer by value (optional)**

Passes the number of halfwords or bytes to be written. A positive value is in halfwords; a negative value is in bytes.

labelid **16-bit signed integer by value (optional)**

Passes the number of the label to be written. The first label is zero. This parameter is ignored for labeled tapes. The next sequential tape label is written. The default is zero.

Operation Notes

Once a disk file is opened, it is possible to read from or write to user-defined labels regardless of the access to the rest of the file.

Condition Codes

CCE	Request granted.
CCG	Request denied. The calling process attempted to write a label beyond the limit specified in the `FOPEN/HPFOPEN` intrinsic when the file was created.
CCL	Request denied. An error occurred.

Refer to this intrinsic in the *MPE/iX Intrinsics Reference Manual* (32650-90028) for other codes pertaining to KSAM files.

HPFOPEN

Establishes access to a file and creates a file.

Syntax

```
                I32   I32      I32V   *
HPFOPEN(filenum,status[,itemnum,item] [...]);
```

Note

Up to 41 *itemnum/item* pairs can be specified.

Parameters

filenum **32-bit signed integer by reference (required)**

Returns a file number used to identify the opened file in subsequent intrinsic calls.

Can be used safely with all file system intrinsics that require a 16-bit file number to be passed in the intrinsic call (for example, FREAD, FWRITE, FCLOSE).

status **32-bit signed integer by reference (optional)**

Returns the status of the HPFOPEN call. If no errors or warnings are encountered, *status* returns 32 bits of zero. If errors or warnings are encountered, *status* is interpreted as two 16-bit fields:

Bits	Value/Meaning
0:16	status.info A negative value indicates an error condition, and a positive value indicates a warning condition.
16:16	status.subsys The value represents the subsystem that set the status information. Refer to the *MPE/iX Error Message Manual Volumes 1, 2 and 3* (32650-90066, 32650-90152, and 32650-90368) for status messages.

Caution

If an error or warning is encountered and the *status* parameter was not specified, HPFOPEN causes the calling process to abort.

itemnum **32-bit signed integer by value (optional)**

Passes the item number, refer to Table 9-6.

item **type varies by reference (optional)**

Passes and/or returns the option indicated by the corresponding *itemnum* parameter, refer to Table 9-6.

Note An *itemnum* takes precedence over any previously specified duplicate *itemnum*. Any duplicated *itemnum* is flagged as a warning.

Table 9-6. HPFOPEN Itemnum/Item Values

Itemnum	Mnemonic	Item Description
0		End of option list: There is no corresponding *item*. The absence of an *itemnum* after the last *itemnum,item* pair is equivalent to specifying this option.

2	CA	Formal designator:
		
		Passes a formal file designator, following MPE/iX file naming conventions. The file name must begin with an alphabetic character and contain alphanumeric characters, slashes, or periods. If the file name is the name of a user-defined file, it can begin with an asterisk (`*`). If the file name is the name of a system-defined file, it can begin with a dollar sign (`$`). Specify the remote location of a device as *filename:envid*. The file, lockword, group, and account names are each limited to eight characters in length. The formal file designator may contain command interpreter variables and expressions that are evaluated by `HPFOPEN` before the formal file designator is parsed and validated.
		
		A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example:
		
		%devname% (% is the delimiter, *devname* is the designator)
		fabcxyzf (*f* is the delimiter, *abcxyz* is the designator)
		
		For a KSAM file, the device must be a random access device such as a disk.
		
		The following are examples of valid MPE/iX formal file designators:
		
		`&file/lock.group.account:node.dest.level&`
		`&filename&`
		`&!myfile&`
		`&!afile/![FINFO("!afile",33)]&`
		
		The following are examples of invalid formal file designators:
		
		`"filename.group` (missing delimiter (”))
		`file.group"` (`'f'` is used as delimiter, missing at end)
		
		Default: A nameless file is assigned that can be read from or written to, but not saved. (The domain of a nameless file must be new.)
		
		Only one of the following options can be in effect when a file is opened:
		
		itemnum=2
		itemnum=51

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
3	I32	Domain: Passes a value indicating which file domain MPE/iX searches to locate the file. A nameless disk file must always be a new file. A device file (such as a tape or terminal) always resides in the system file domain (permanent file directory). Always specify a device file as old or permanent. The following values are valid: 0 The file is a new temporary file. It is not placed in a directory. 1 The file is a permanent file, found in the system file domain. 2 The file is a temporary file, found in the job file domain. 3 The file is an old (permanent or temporary) file. The job file domain is searched first. If the file is not found, the system file domain is searched. 4 The file is created, placed in the permanent file directory, and becomes a permanent file. Default: 0
5	I32	Designator: Passes a value indicating a special file opening. Any of the following special files can be specified with the *itemnum*=2. For example, a file name of `$STDLIST` opens the standard list device. The following values are valid: 0 Allows all other options to specify the file. 1 The actual file designator is `$STDLIST`. 2 The actual file designator is `$NEWPASS`. 3 The actual file designator is `$OLDPASS`. 4 The actual file designator is `$STDIN`. 5 The actual file designator is `$STDINX`. 6 The actual file designator is `$NULL`. Default: 0 For example, passing `&MYFILE&` in *itemnum*=2 and using *itemnum*=5 and *item*=4 to equate it with `$STDIN` is equivalent to the file equation `FILE MYFILE=$STDIN`. This option is not equated with *itemnum*=2 if both of the following conditions are true: ■ The *itemnum*=9 option allows file equations for the file opening. ■ An explicit or implicit `FILE` command equating the formal file designator to a different actual file designator occurs in the job or session. A leading `*` in a formal file designator passed by *itemnum*=2 overrides an *itemnum*=9 option.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
6	I32	Record format: Passes a value indicating the internal record structure desired for the file. This option is applicable only at file creation. Only a fixed-length record is allowed for KSAM XL files (0). Default: 0
9	I32	Disallow file equation: Passes a value indicating whether or not MPE/iX file equations are allowed. A leading `*` in a formal file designator overrides the setting to disallow `FILE` equations. The following values are valid: 0 Allow `FILE` equations to override programmatic or system-defined file specifications. 1 Disallow `FILE` equations from overriding programmatic or system-defined file specifications. Default: 0
10	I32	File type: Passes a value indicating the internal record structure used to access records in the file. If the file is old, this option is ignored. Specifying an *itemnum*=5 value other than zero overrides this option. This option is applicable only at file creation. The following values are valid: 0 Standard (STD) file 1 KSAM/3000 file 2 Relative I/O (RIO) file 3 KSAM XL file 4 Circular (CIR) file 6 Message (MSG) file Default: 0

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
11	I32	Access type: Passes a value indicating the type of access intended for the file. This option restricts usage of the file system intrinsics. The following values are valid: 0 Read access only, if the file's security provisions allow read access. **FWRITE**, **FUPDATE**, and **FREMOVE** intrinsic calls cannot reference this file. The end-of-file (EOF) is not changed. (Default) 1 Write access only, if the file's security provisions allow write access. Any data written in the file prior to the current **HPFOPEN** request is deleted. **FFINDBYKEY**, **FFINDN**, **FPOINT**, **FREAD**, **FREADBYKEY**, **FREADC**, **FREADDIR**, **FREMOVE**, **FSPACE**, and **FUPDATE** intrinsic calls cannot reference this file. The EOF is set to zero. 2 Write-save access only, if the file's security provisions allow write access. Previous data in the file is not deleted. **FFINDBYKEY**, **FFINDN**, **FPOINT**, **FREAD**, **FREADBYKEY**, **FREADC**, **FREADDIR**, **FREMOVE**, **FSPACE**, and **FUPDATE** intrinsic calls cannot reference this file. The EOF is not changed. Therefore, data is overwritten if **FWRITE** is called. The system changes this value to append for message files. 3 Append access only, if the file's security provisions allow either append or write access. **FFINDBYKEY**, **FFINDN**, **FPOINT**, **FREAD**, **FREADBYKEY**, **FREADC**, **FREADDIR**, **FREMOVE**, **FSPACE**, and **FUPDATE** intrinsic calls cannot reference this file. The record pointer is set to EOF prior to each **FWRITE**. For disk files, the EOF is updated after each **FWRITE** call. Therefore, data cannot be overwritten. 4 Read/write (I/O) access only, if the file's security provisions allow both read and write access. If both read and write access are not allowed, the access type is limited to that specified in the security provisions (either read or write). Any file intrinsic except **FUPDATE** and **FREMOVE** can be called for this file. The EOF is not changed. This option is not valid for message files. 5 Update access only, if the file's security provisions allow both read and write access. If both read and write access are not allowed, the access type is limited to that specified in the security provisions (either read or write). All file intrinsics can be called for this file. The EOF is not changed. This option is not valid for message files. 6 Execute access only, if the file's security provisions allow execute access. This allows read/write access to any loaded file. The program must be running in privileged mode to specify execute access. This option is not valid for message files. 7 Execute-read access only, if the file's security provisions allow execute access. This allows only read access to a loaded file. The program must be running in PM to specify execute-read access. This is changed to execute access for KSAM, CIR, and RIO files. Not valid for message files.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
12	I32	Dynamic locking: Passes a value enabling or disabling file locking for the file. When specified, the FLOCK and FUNLOCK intrinsics can be used to dynamically permit or restrict concurrent access to a disk file by other processes at specified times. The following values are valid: 0 Disallow dynamic locking/unlocking 1 Allow dynamic locking/unlocking Default: 0 The process can continue this temporary locking or unlocking until it closes the file. If several accessors are sharing the file, they must all specify, or not specify, this option. For example, if a file is opened with the dynamic locking option enabled, and a subsequent accessor tries to open the file with dynamic locking disabled, that subsequent attempt to open fails. Dynamic locking and unlocking are possible through the equivalent of a global resource identification number (RIN) assigned to the file and temporarily acquired by HPFOPEN. Accessors that have opened a file with the dynamic locking option enabled must access the file through the FLOCK and FUNLOCK intrinsics to ensure exclusive use of the file. These accessors are allowed concurrent access even when not using FLOCK and FUNLOCK, but exclusive access is not guaranteed. Note: The file system does not guarantee exclusive access, even when FLOCK and FUNLOCK are used, unless all programs that access the file cooperate by using locking. A program that opens the file with dynamic locking enabled will still be allowd to modify the file, even if it never calls FLOCK. Lock access must be at the account, group, and file levels for HPFOPEN to grant this option. (Lock access is available if lock, execute, append, or write access is set at these levels.) This option is ignored for files not residing on disk.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
13	I32	Exclusive: Passes a value indicating continuous exclusive access to the file, from open to close. Use this option when performing a critical operation (for example, updating the file). The following values are valid: 0 If *itemnum*=11 specifies read only access, read-share access takes effect. Otherwise, exclusive access takes effect. Regardless of which access option was selected, `FFILEINFO` reports zero. 1 Exclusive access. After the file is opened, any additional `HPFOPEN/FOPEN` requests for this file, whether issued by this process or another process, are prohibited until this process issues the `FCLOSE` request or terminates. If any process is already accessing this file when an `HPFOPEN/FOPEN` call is issued with exclusive access specified, an error status is returned to the process. If another `HPFOPEN/FOPEN` call is issued for this file while exclusive access is in effect, an error code is returned to the process that issued that `HPFOPEN/FOPEN` call. Request exclusive access only if the lock access mode is allowed by the security provisions for the file. For message files, specifying this value means that there can be only one reader and one writer. 2 Read-share access (semi-exclusive access). After the file is opened, concurrent write access to this file through another `HPFOPEN/FOPEN` request is prohibited, whether issued by this process or another process, until this process issues the `FCLOSE` request or terminates. A subsequent request for the read/write or update *itemnum*=11 obtains read access. However, other types of read access are allowed. If a process already has write access to the file when this `HPFOPEN` call is issued, an error code is returned to the calling process. If another `HPFOPEN/FOPEN` call that violates the read-only restriction is issued while read-share access is in effect, that call fails and an error code is returned to the calling process. You can request read-share access only if you are allowed the lock access mode by the security provisions for the file. For message files, specifying this value means that there can be multiple readers, but only one writer. 3 Share access. After the file is opened, this permits concurrent access to this file by any process, in any access mode, subject to other basic MPE/iX security provisions in effect. For message files, specifying this value means that there can be multiple readers and multiple writers. Default: 0

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
17	I32	Copy mode: Passes a value that determines if any file should be treated as a standard sequential file so it can be copied by logical record or physical block to another file. The following values are valid: 0 The file is accessed as its own file type (for example, a message file is treated as a message file). 1 The file is to be treated as a standard (STD) file, with variable-length records. For message files, this allows nondestructive reading of an old message file at either the logical record or physical block record level. Only block-level access is permitted if the file is opened with write access. This prevents incorrectly formatted data from being written to the message file while it is unprotected. To access a message file in copy mode, a process must have exclusive access to the file. Default: 0
18	@32	Short-mapped: Returns a short pointer to the beginning of the data area of the file. This option maps the file into short pointer space. A short-mapped file can be 4-megabytes in length. The calling process can have up to 6-megabytes of short mapped files open at a time. Use the pointer as a large array of any type to efficiently access the file. A file previously opened normally (not mapped) or with the long-mapped option is not accessible with the short-mapped option. If this option is specified with the file already opened into long pointer space, an error results. A loaded program file or a loaded library file is not accessible with the short-mapped option. A file cannot be loaded that is currently opened with the short-mapped option. Sharing of short pointer files is provided through normal file system sharing mechanisms, for example, use of the exclusive option. With the short-mapped file, all file system intrinsics, applicable to the file, can be used. `FREAD` and `FWRITE` calls can be mixed with the short-mapped access. Standard (`STD`) type disk files of fixed or undefined record length can be accessed short-mapped with the access type option set to any value. Standard type disk files of variable record length can be accessed short-mapped only if the access type option is set to read-only access. KSAM files can be accessed short-mapped only if the access type option is set read-only access and the copy mode option is set to 1. Default: No short pointer returned

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
19	I32	Record size: Passes the size, in bytes, of the logical records in the file. Valid range is dependent upon both storage format (ASCII or binary) and record format. For fixed-length and undefined-length ASCII files, a record size can be specified in the range 1 to 32,767. For variable-length ASCII files, and for fixed-length, variable-length, and undefined-length binary files, a record size can be specified in the range 1 to 32,766. `HPFOPEN` rounds up odd values to the next highest even number (equivalent to the nearest halfword boundary) if the file is ASCII with variable-length record format, or binary with fixed-length, variable-length, or undefined-length record format. For example, if a record size of 105 is specified for a fixed-length binary file, `HPFOPEN` sets the record size to 106; if a record size of 233 is specified for a fixed-length ASCII file, the record size remains the same as it was when specified. Default: 256
20	CA	Device name: Passes the logical device number, in ASCII form, of a specific device. The file is assumed to be permanent. If the device name option is specified, the nonshareable device should be ready prior to the `HPFOPEN` call (otherwise, an error results). Only one of the following options can be in effect when a file is opened: *itemnum*=20 *itemnum*=22 *itemnum*=23 *itemnum*=42 Default: disk file located on the volume class `disc` associated with the group in which file resides. A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%devname%* (% is the delimiter, *devname* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator) For a KSAM file, the device must be a random access device such as a disk.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
22	CA	Volume class: Passes a character array representing a volume class name where the file space is to be restricted. This option is applicable only at file creation. A volume class is a subset of volumes within a volume set. The volume class name must be a valid volume class name residing on the volume set bound to the volume (the volume set is an attribute of the group in which the file resides). Only one of the following options can be in effect when a file is opened with this option: *itemnum*=20 *itemnum*=22 *itemnum*=23 *itemnum*=42 Default: A disk file located on the volume class `DISC` associated with the group in which the file resides. A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%volclass%* (% is the delimiter, *volclass* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator)
23	CA	Volume name: Passes a character array representing a volume name that restricts the file specified to a specific volume. The volume must reside within the volume set of the group where the file resides. This option is applicable only at file creation. Only one of the following options can be in effect when a file is opened with this option: *itemnum*=20 *itemnum*=22 *itemnum*=23 *itemnum*=42 Default: A disk file located on the volume class `DISC` associated with the group in which the file resides. A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%volclass%* (% is the delimiter, *volclass* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator)

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
26	CA	Remote environment: Passes the node name of the remote computer where the file is located. This option is used when referencing a file located on a remote computer. Default: No node name passed (local file access) A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%envname%* (% is the delimiter, *envname* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator)
29	I32	Privileged access: Passes a value that temporarily restricts access to the file number returned from `HPFOPEN` to a calling process whose execution level is equal to or less than the value specified in this option. This restriction lasts until the file associated with the restricted file number is closed. Do not specify a value less than the execution level of the calling process. The following values are valid: 0 Privilege level zero (most privileged level) 1 Privilege level one 2 Privilege level two 3 Privilege level three (least privileged level) Default: The execution level of the calling process
33	I32	User labels: Passes the number, in the range 0 to 254, of user-label records to be created for the file. Applicable for new disk files only. Default: 0
35	I32	File size: Passes the maximum file capacity: ■ For variable-length records, the capacity is expressed in blocks (*blockitem#=recordsize * blockfactor*). ■ For fixed-length and undefined-length records, the capacity is expressed in logical records. ■ The maximum file size for standard and KSAM files is 2-gigabytes. ■ The maximum file size of 500-megabytes, for RIO, circular, and message files, is dependent upon both the record size and the number of extents defined for the file: □ For circular and RIO files, *recsize*=256 bytes and *numextent*=32. □ For message files, *recsize*=128 bytes and *numextent*=32. This option is applicable only at file creation. Default: 2-gigabytes

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
36	I32	Initial allocation: Passes a positive integer value indicating the number of extents to be allocated to the file initially. This option is applicable only at file creation. Default: 0
37	I32	Filecode: Passes a value that can be used as a file code to identify the type of file. This code is recorded in the file label and is accessible through the `FFILEINFO` intrinsic. This option is applicable only at file creation (except when opening an old file that has a negative file code). If the program is running in user mode, specify a file code in the range 0 to 32,767 to indicate the file type being created. Programs running in user mode can access files with positive file codes only. If the program is running in privileged mode, specify a file code in the range -32,768 to 32,767. Programs running in privileged mode can access files with a file code in the range -32,768 to 32,767. If an old file is opened that has a negative file code in its file label, the file code specified must match the file code in the file label (otherwise, an error results). Default: 0
38	I32	File privilege: Passes a value that determines a permanent privilege level to be associated with a newly created file. This option permanently restricts file access to a process whose execution level is less than or equal to the specified value. A value cannot be specified for less than the execution level of the calling process. This option is applicable only at file creation. The following values are valid: 0 Privilege level zero (most privileged level) 1 Privilege level one 2 Privilege level two 3 Privilege level three (least privileged level) Default: 3 A file created with levels 0, 1, or 2 can be opened only with the `HPFOPEN` intrinsic; the `FOPEN` intrinsic cannot be used.
41		Reserved for MPE/iX.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
42	CA	Device class: Passes a device class where the file will reside. The file system uses the device class name to select a nonshareable device from a configured list of available devices. The name can have a length of up to eight alphanumeric characters, beginning with a letter (for example, `TAPE`). If a device class is specified, the file is allocated to any available device in that class. Only one of the following options can be in effect when a file is opened: *itemnum*=20 *itemnum*=22 *itemnum*=23 *itemnum*=42 Default: A disk file located on the volume class `DISC` associated with the group in which the file resides. A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%devclass%* (% is the delimiter, *devclass* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator)
43	record	UFID: Passes a unique file identifier (UFID) to provide a fast opening of an old disk file. A UFID is a record structure, 20 bytes in length, that uniquely identifies a disk file. Using this option avoids a directory search. Obtain the UFID of an opened file by calling `FFILEINFO`. The UFID can then be passed to `HPFOPEN`. The file represented by the UFID must be accessible to the process calling `HPFOPEN` (all file system security checks are made). New files cannot be opened with this option. If the file to be opened by the UFID contains a lockword, use *itemnum*=2 to specify the file name with the lockword. Default: No UFID passed (a directory search is performed)
45	CA	Fill character: Passes two ASCII characters that determine what padding character to use at the end of blocks or unused pages, and the padding used by *itemnum*=53. Do not use delimiter characters for this option. The fill character must be a 2-byte array. The first character only is used as the padding character. The second character is reserved for future use. This option is applicable only at file creation. Default: Null characters for a binary file and ASCII blanks for an ASCII file.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
47	I32	Numextents: Passes a value in the range 1 to 32 that determines the number of extents for the file. This parameter is kept mainly for compatibility with MPE/V. Its main usefulness is that a file may be created with 1 contiguous extent. If a value of 1 is specified, the file is created as one contiguous extent of disk space. If a value greater than 1 is specified, a variable number of extents (with varying extent sizes) is allocated on a need basis. This option is applicable only at file creation. To get one initially allocated continuous extent, specify both numextent=1 and initialloc=1. Default: 1
49		Reserved for MPE/iX.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
50	I32	Final disposition: Passes a value indicating the final disposition of the file at close time (significant only for files on disk and magnetic tape). A corresponding parameter in a `FILE` command can override this option, unless file equations are disallowed with *itemnum*=9. The following values are valid: 0 No change. The disposition remains as it was before the file was opened. If the file is new, it is deleted by `FCLOSE`; otherwise, the file is assigned to the domain it belonged to previously. An unlabeled tape file is rewound. If the file resides on a labeled tape, the tape is rewound and unloaded. 1 Permanent file. If the file is a disk file, it is saved in the system file domain. A new or temporary file on disk has an entry created for it in the system (permanent) file directory. Should a file of the same name already exist in the directory, an error code is returned at close time and the file remains open. If the file is a permanent file on disk, this domain disposition has no effect. Also, if the file is stored on magnetic tape, the tape is rewound and unloaded. 2 Temporary job file (rewound). The file is retained in your temporary (job or session) file domain and can be requested by any process within your job or session. If the file is a disk file, the uniqueness of the file name is checked. Should a file of the same name already exist in the temporary file domain, an error code is returned at close time and the file remains open. When a file resides on unlabeled magnetic tape, the tape is rewound. However, if the file resides on labeled magnetic tape, the tape is backspaced to the beginning of the presently opened file. 3 Temporary job file (not rewound). This value has the same effect as specifying final disposition option, except that tape files are not rewound. In the case of unlabeled magnetic tape, if the `FCLOSE` is the last done on the device (with no other `FOPEN/HPFOPEN` calls outstanding), the tape is rewound and unloaded. If the file resides on a labeled magnetic tape, the tape is positioned to the beginning of the next file on the tape. 4 Released file. The file is deleted from the system. 5 Convert a permanent file to a temporary file. The file is removed from the permanent file directory and placed in the temporary file directory. (Privileged mode capability is required to use this option.) Default: 0 For more information on file disposition at close time, refer to the description of the `FCLOSE` intrinsic.

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
51		Pascal XL string: Passes a formal file designator, following MPE/iX file naming conventions, but using the Pascal/iX STRING type format. This option is identical to *itemnum*=2 except for the type of item. No delimiters are needed. Default: No string passed Only one of the following options can be in effect when a file is opened: *itemnum*=2 *itemnum*=51
52	CA	File equation string: Passes a character string that matches the MPE/iX file equation specification syntax exactly. This option allows the specification of options available in the `FILE` command. The *formaldesig* parameter and *filereference* parameter can contain embedded command interpreter variables and expressions. However, there cannot be more than eight characters in each of these components (*filename, lockword, groupname, accountname*) including the command interpreter variable and expression characters. Default: No string passed A character placed in the first element designates the delimiter used by `HPFOPEN` to search for the end of the character array. The delimiter can appear again only following the last valid character of the character array, for example: *%fileequation%* (% is the delimiter, *fileequation* is the designator) *fabcxyzf* (*f* is the delimiter, *abcxyz* is the designator)
53	I32	ASCII/binary: Passes a value indicating whether ASCII or binary code is to be used for a new file when it is written to a device that supports both codes. For disk files, this may affect padding that can occur when issuing a direct-write intrinsic call (`FWRITEDIR`) to a record that lies beyond the current logical end-of-file indicator. The fill character is specified during the file creation. Default for ASCII is blank. Default for binary is binary 0. By default, magnetic tape and files are treated as ASCII files. This option is applicable only at file creation. The following values are valid: 0 Binary file 1 ASCII file Default: 0

Table 9-6. HPFOPEN Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
54	REC	KSAM parm: Passes a record that defines the keys for a new KSAM file. The format of the parameter is the same as the `FOPEN` intrinsic *ksamparam* field. Default: No record passed
55		Reserved for MPE/iX
56	I32	Object class: Passes a user object class number, in the range 0 to 10, that is associated with the file. Default: Determined by the file code for system and subsystem files, and by the file type and record type for normal user files.
57		Reserved for MPE/iX.
58		Reserved for MPE/iX.
59		Reserved for MPE/iX.
60		Reserved for MPE/iX.
61		Reserved for MPE/iX.
64		ACD.

Table 9-7. FOPEN/HPFOPEN Parameter Equivalents

FOPEN Parameter	HPFOPEN Itemnum,Item
filenum (functional return)	*filenum* (parameter)
formaldesig	*2,formaldesig*
foption: Bits (14:2) Domain Bit (13:1) ASCII/binary Bits (10:3) File designator Bits (8:2) Record format Bit (7:1) Carriage-control Bit (6:1) Labeled tape Bit (5:1) Disallow file equation Bits (2:3) File type	 *3, domain* *53, ASCII/binary* *5, file designator* *6, record format* *7, carriage-control* *8, labeled tape* *9, disallow file equation* *10, file type*
aoption: Bits (12:4) Access type Bit (11:1) Multirecord Bit (10:1) Dynamic locking Bits (8:2) Exclusive Bit (7:1) Inhibit buffering Bits (5:2) Multiaccess mode Bit (4:1) Nowait I/O Bit (3:1) File copy	 *11, access type* *15, multirecord* *12, dynamic locking* *13, exclusive* *46, inhibit buffering* *14, multiaccess mode* *16, nowait I/O* *17, file copy*
recsize	*19, record size*
device	*20, device name* *22, volume class* *23, volume name* *24, density* *25, printer environment* *26, remote environment* *42, device class* *48, reverse VT*
formmsg	*8, labeled tape label* *28, spooled message* *30, labeled tape type* *31, labeled tape expiration* *32, labeled tape sequence* *54, KSAM parms*
userlabels	*33, user labels*
blockfactor	*40, block factor*

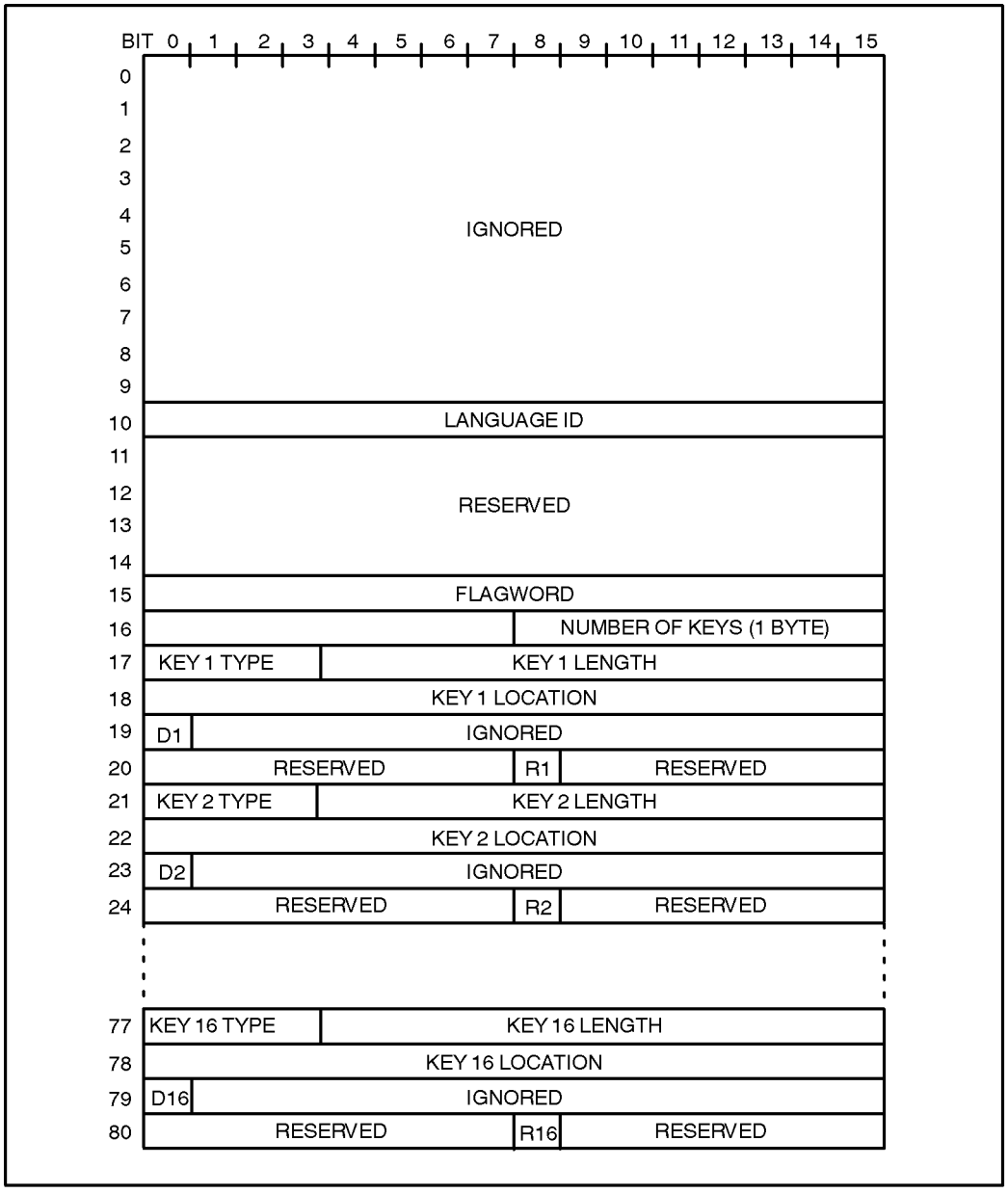
Table 9-7.
FOPEN/HPFOPEN Parameter Equivalents (continued)

FOPEN Parameter	HPFOPEN Itemnum,Item
numbuffers: Bits (11:5) Numbuffers Bits (4:7) Spooler copies Bits (0:4) Output priority	 *44, numbuffers* *34, spooler copies* *27, output priority*
filesize	*35, filesize*
numextent	*47, numextent*
initialloc	*36, initial allocation*
filecode	*37, filecode*

Operation Notes

Enables creation of a new file on a shareable device and defines the physical characteristics of that file prior to access. Enables access to existing files. Returns a file number to the calling process that uniquely identifies the file. Use the file number to reference the file in calls to other intrinsics.

The format of the KSAM parameter is shown in Figure 9-7.



LG200166_004

Figure 9-7. HPFOPEN KSAM XL Parameter Format

A

COBOL Intrinsics

COBOL compilers (COBOL 68 and earlier) required special intrinsics to access keyed files. The following intrinsics are provided only for the maintenance of COBOL 68 or earlier COBOL programs using KSAM structures.

Note

Do not use these intrinsics for new programming. Current COBOL file access modules provide KSAM file access.

Calling a KSAM Procedure

KSAM files are accessed from COBOL programs through calls to a set of procedures. These procedures allow you to open, open for shared access, write records to, read records from, lock, unlock, update, position, and close a KSAM file. The COBOL procedures provided with KSAM/3000 correspond to the INDEXED I/O module statements in COBOL 74.

In HP COBOL/3000, the procedures that are used to access KSAM files differ in form from the COBOL input/output statements used to access non-KSAM files. The KSAM interface procedures use parameters for information that would otherwise be specified in the FILE-CONTROL paragraph and the FD entry of the DATA DIVISION. These parameters are themselves defined in the WORKING-STORAGE section of the DATA DIVISION. The main restriction on the KSAM interface call parameters is that they must be 16 bit aligned.

The KSAM interface procedures are called using a `CALL` statement of the following general form.

```
CALL "name" USING filetable,status [,parameter[, ... ] ]
```

Where:

"name" identifies the procedure to which control is transferred.

filetable an 8-halfword table that identifies the file by name and in which access mode and input/output type are specified, and to which is returned the file number on open, and a code identifying the previous operation.

status One halfword to which a two-character code is returned that indicates the status of the

input/output operation performed on the file by the called procedure.

parameter — One or more parameters, depending on the particular procedure called, that further define operations to be performed on the file.

The first two parameters, *filetable* and *status*, are included in every KSAM procedure call except `CKERROR`; other parameters may be specified depending on the particular procedure. If a parameter is included in the procedure format, then it must be included in the procedure call. All parameters are required.

Another characteristic of KSAM procedure call parameters is that they must always start on a halfword boundary. In order to ensure this, the parameters should be defined in the WORKING-STORAGE SECTION as 01 record items, 77 level elementary items, or else the SYNCHRONIZED clause should be included in their definition.

A literal value cannot be used as a parameter to these procedures. Any value assigned to a data item used as a parameter is passed to the procedure, but a literal value causes an error.

Depending on the procedure, certain data items may be assigned values as a result of executing the procedure.

Note

There are no COBOL procedures to read a KSAM file in physical order or to access a record by its physical record number. (Physical order is the order in which the data records were written to the file.)

Filetable Parameter

The first parameter in every KSAM procedure call must be *filetable*, a table describing the file and its access. This table is defined in the WORKING-STORAGE SECTION of the COBOL program. It requires eight halfwords as illustrated in Figure A-1.

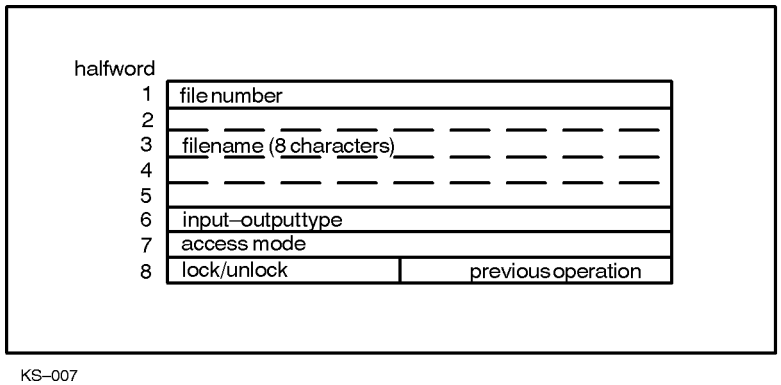


Figure A-1. Filetable Structure

filenumber A number identifying the file returned by the `CKOPEN` procedure after the file named in halfwords 2-5 has been successfully opened. After the file is closed by `CKCLOSE`, filenumber is reset to 0. (This number should be set to zero when the file table is initially defined.) It must be defined as a COMPUTATIONAL item.

filename The name of the KSAM file. This name is the actual designator assigned to the file when it is created with the KSAMUTIL or MPE/iX `BUILD` command; filename may be a formal designator if it is equated to the actual designator in a `FILE` command.

input/output type A code that limits the file access to input only, output only, or allows both input and output:

0 input only

1 output only

2 input/output

It must be defined as a COMPUTATIONAL item.

access mode A code that indicates how the file will be processed: sequentially only, randomly only, or either (dynamically):

0 sequential only

1 random only

2 dynamic (sequential or random)

It must be defined as a COMPUTATIONAL item.

previous operation A code in the right byte of halfword 8 of the file table indicating the previous successful operation:

0 previous operation unsuccessful or there has been no previous operation on this file

1 `CKOPEN` successful

2 `CKSTART` successful

3 `CKREAD` successful

4 `CKREADBYKEY` successful

5 `CKDELETE` successful

6 `CKWRITE` successful

7 `CKREWRITE` successful

8 `CKCLOSE` successful

9 `CKOPENSHR` successful

This field should be set to zero when the file table is initially defined and thereafter should not be

altered by the programmer. It must be defined as a COMPUTATIONAL item.

lock/unlock A code in the left byte of halfword 8 of the file table that indicates whether a CKLOCK or CKUNLOCK has been performed successfully since the operation specified in previous operation:

10 CKLOCK successful

11 CKUNLOCK successful

A sample file table definition might be:

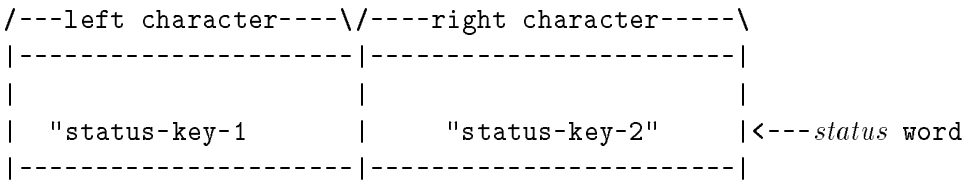
```
WORKING-STORAGE SECTION.
  FILE_TABLE.
    01 KSAM_FILE.
      02 FILENUMBER    PIC S9(4) COMP VALUE 0.
      02 FILENAME      PIC X(8) VALUE "KSAMFILE".
      02 I-O-TYPE      PIC S9(4) COMP VALUE 0.
      02 A-MODE        PIC S9(4) COMP VALUE 0.
      02 PREV-OP       PIC S9(4) COMP VALUE 0.
```

The file table identifies a file created with the name KSAMFILE as a file to be opened for sequential input only. The values of I-O-TYPE and A-MODE can be changed following a call to CKCLOSE for the file.

Status Parameter

The *status* parameter is a two-character item to which the status of the input/output operation is returned. It is always the second parameter in a KSAM procedure call. The *status* parameter must be defined in the WORKING-STORAGE SECTION of the COBOL program.

Status consists of two separate characters: the left character is known as status-key-1, and the right is known as status-key-2.



Combining status-key-1 with status-key-2, the following values may be returned to the *status* parameter as a whole:

00 *Successful completion*—

The current input/output operation was completed successfully; no duplicate keys were read or written.

02 *Successful completion*; *Duplicate key*—

For a `CKREAD` or a `CKREADBYKEY` call, the current alternate key has the same value as the equivalent key in the sequentially following record; duplicate keys are allowed for the key. For a `CKWRITE` or `CKREWRITE` call, the record just written created a duplicate key value for at least one alternate key for which duplicates are allowed.

10 *At End condition*—

In a sequential read using `CKREAD`, no next logical record was in the file.

21 *Invalid key; Sequence error*—

A call to `CKWRITE` attempted to write a record with a key that is not in sequentially ascending order, to a file opened for sequential access.

A call to `CKREWRITE` was attempted but the primary key value was changed by the program since the previous successful call to `CKREAD`.

22 *Invalid key; Duplicate key*—

An attempt was made to write or rewrite a record with `CKWRITE` or `CKREWRITE` and the record would create a duplicate key value for a key where duplicates are prohibited.

23 *Invalid key; No record found*—

An attempt was made with `CKSTART` or `CKREADBYKEY` to access a record identified by key, but no record is found with the specified key value at the specified location.

24 *Invalid key; Boundary violation*—

An attempt was made with a call to `CKWRITE` to write past the externally defined boundaries of the file; that is, to write past the end-of-file.

30 *Lock denied*—

An attempt was made to lock a file already locked by another process; or file was not opened with dynamic locking allowed.

31 *Unlock denied*—

An attempt was made to unlock a file with `CKUNLOCK`, but the file had not been locked by `CKLOCK`.

9*n* *File system error*—

A call to an input/output procedure was unsuccessful as a result of a file system error, not one of the error conditions defined for the other *status* values. The value of status-key-2 (*n*) is a binary number between 0 and 255 that corresponds to an MPE file system error code. To convert this binary value to numeric display format, call the `CKERROR` routine.

The value of *status* can be tested as a whole, or the two characters can be tested separately as *status-key-1* and *status-key-2*. In any case, the status of each call should be tested immediately following execution of the call. Unless the first character of *status* = 0, the call was not successful.

For example, a sample *status* parameter definition might be:

```
WORKING-STORAGE SECTION.
⋮
01 STAT.
     02 STATUS-KEY-1 PIC X.
     02 STATUS-KEY-2 PIC X.
```

These items can then be referenced in the PROCEDURE DIVISION. For example: to test only the first character:

```
IF STATUS-KEY-1 NOT = "0" THEN
    GO TO "ERROR-ROUTINE".
```

To test the entire status word:

```
IF STAT = "23" THEN
     DISPLAY "RECORD NOT FOUND".
```

Note that the word *STATUS* is reserved.

KSAM Logical Record Pointer

Many of the KSAM procedures use a *logical record pointer* to indicate the current record in the file. This pointer points to a key value in the index area that identifies the current record in the data area. The particular key used, if the file has more than one key, is the key specified in the current procedure or the last procedure that referenced a key.

Procedures that use pointers are either *pointer-dependent* or *pointer-independent*. Pointer-dependent procedures expect the pointer to be positioned at a particular record in order to execute correctly. Pointer-independent procedures, on the other hand, execute regardless of where the pointer is positioned and, in most cases, they position the pointer.

Table A-1. Positioning the Logical Record Pointer

Procedure Name	Pointer-Dependent	Position of Pointer After Execution of Procedure
CKSTART	NO	Points to key whose value was specified in call.
CKREADBYKEY	NO	Points to key whose value was specified in call.
CKWRITE	NO	Points to key whose value is next in key sequence to key value in record just written.
CKREAD	YES	Pointer remains positioned to key value for record just read; *unless* next call is to CKREAD, or to CKREWRITE followed by CKREAD, in which case, next CKREAD moves pointer to next key in key sequence before reading the record.
CKDELETE	YES	Points to next key value in ascending sequence following key value in record just deleted.
CKREWRITE	YES (sequential mode) NO (random or dynamic mode)	Pointer remains positioned to key value for record just modified, unless any key value in record was changed; in this case, it points to next key in ascending sequence after the key in the modified record.

Shared Access

Particular care must be taken when using the logical record pointer during shared access (the file was opened with CKOPENSHR). If more than one user opens the same file, one user may modify the record pointer. This causes other users to access the data record.

To avoid this problem, you should always lock the file in a shared environment before calling a procedure that sets the pointer and leave the file locked until all procedures that depend on the pointer have been executed. Thus, if you want to read the file sequentially, delete a record, or modify a record, you should lock the file, call a procedure that sets the pointer (such as CKSTART), and then call CKREAD, CKDELETE, or CKREWRITE. When the operation is complete, you can then unlock the file to give other users access to it.

Sample KSAM File

The file `KSAMFILE` illustrated in Figure A-2 is used in all subsequent examples associated with the COBOL procedure calls.

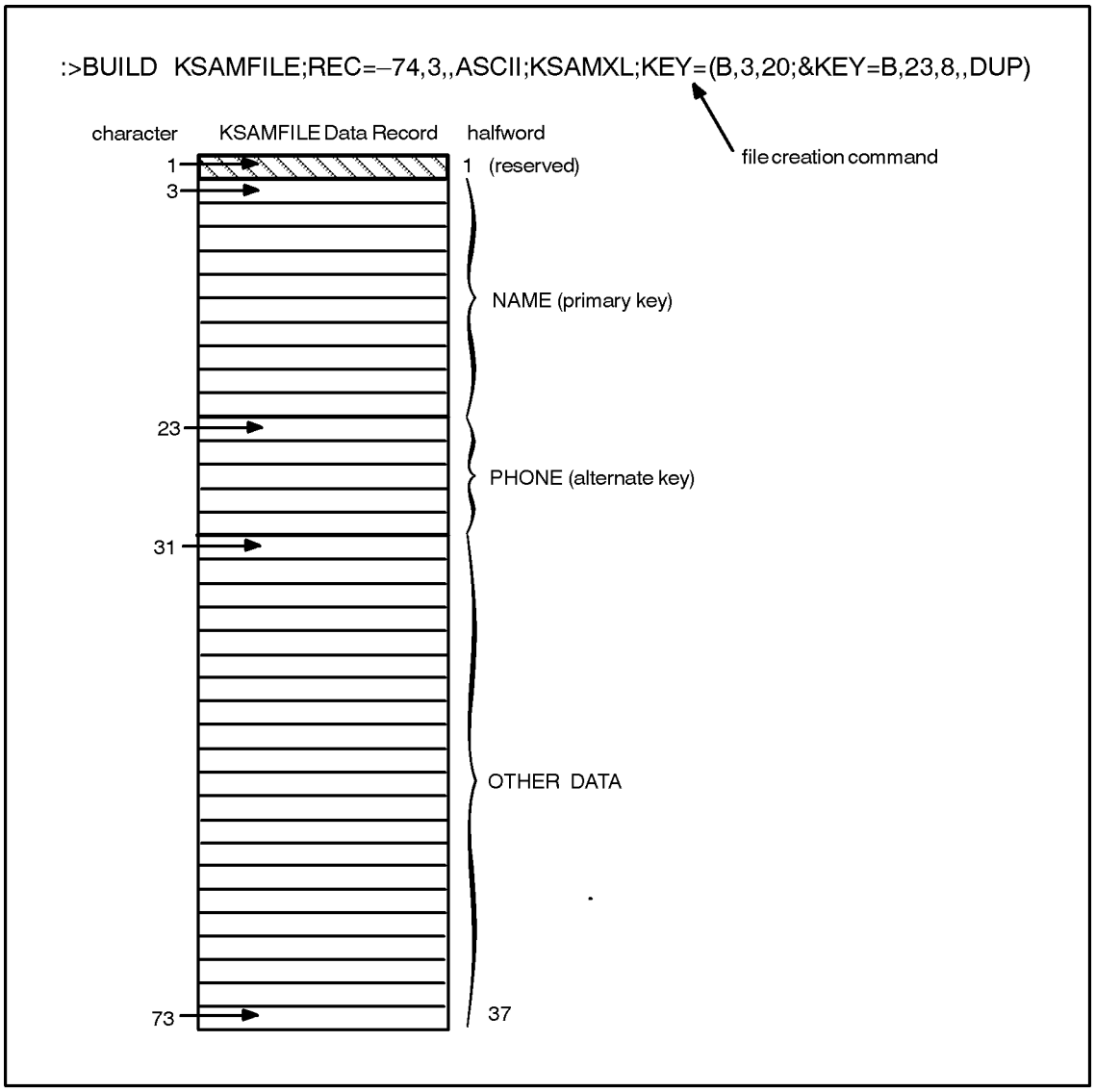


Figure A-2. Representation of KSAMFILE Used in COBOL Examples

A File Description in Working Storage for Figure A-2 appears on the following page.

File Description in Working Storage (Figure A-2).

```
WORKING-STORAGE SECTION
77  RECSIZE  PIC S9(4) COMP VALUE 74.
77  RESULT  PIC 9(4) VALUE 0.
01  REC.
    03  FILLER  PIC XX  VALUE SPACES.
    03  NAME  PIC X(20).
    03  PHONE  PIC X(8).
    03  OTHERDATA PIC X(44).
01  DAT.
    03  NAME  PIC X(20).
    03  PHONE  PIC X(8).
    03  OTHERDATA PIC X(44).
01  FILETABLE.
    03  FILETABLE PIC S9(4) COMP VALUE 0.
    03  FILENAME PIC X(8) VALUE "KSAMFILE".
    03  I-O-TYPE PIC S9(4) COMP VALUE 0.
    03  A-MODE  PIC S9(4) COMP VALUE 0.
    03  PREV-OP  PIC S9(4) COMP VALUE 0.
01  STAT.
    03  STATUS-KEY-1 PIC X.
    03  STATUS-KEY-2 PIC X.
```

CKCLOSE

A call to CKCLOSE terminates file processing for the specified KSAM file.

```
CALL "CKCLOSE" USING filetable, status
```

When processing is completed, a KSAM file should be closed with a call to CKCLOSE. No further processing is allowed on the file until a CKOPEN procedure call opens the file.

CKCLOSE can be executed only for a file that is open.

Parameters

filetable — An 8 halfword record containing: the name of the file, its input/output type, access mode, the file number given the file when it was last opened, and a code indicating whether the previous operation on the file was successful and if so what it was. (Refer to Filetable Parameter discussion earlier in this section.)

status — One-halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKCLOSE. It indicates whether or not the file was successfully closed and if not, why not. The left character is set to 0 if CKCLOSE is successful, to 9 if not. The right character is set to 0 if CKCLOSE is successful, to the file system error code if not. (Refer to Status Parameter discussion earlier in this section.)

Operation Notes

Upon successful completion of CKCLOSE, the file identified by *filetable* is no longer available for processing. Note that a KSAM file can be closed and then reopened in order to specify a different access mode or input/output type.

```
FINISH.
  CALL "CKCLOSE" USING FILETABLE, STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKCLOSE ERROR NO. ", RESULT;
  ELSE DISPLAY "CKCLOSE SUCCESSFUL".
```

CKDELETE

This procedure logically deletes a record from a KSAM file.

```
CALL "CKDELETE" USING filetable, status
```

In order to logically delete records from a KSAM file, you can use the procedure CKDELETE. If reuse is not specified, then a logically deleted record is marked for deletion, but is not physically removed from the file. The deletion mark makes such a record inaccessible but does not physically reduce the size of the file. The utility program FCOPY can be used to compact a KSAM file by copying only active records, excluding deleted records, to a new KSAM file.

CKDELETE deletes the record at which the logical record pointer is currently positioned. Therefore, CKDELETE must be preceded by a call that positions the pointer.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so what it was. (Refer to Filetable Parameter discussion earlier in this section.)

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKDELETE indicating whether the call was successful and if not, why not. (Refer to Status Parameter discussion earlier in this section.)

Operation Notes

In order to delete a record, you should first read the record into the working storage section of your program with a call to CKREAD if in sequential mode, a call to CKREADBYKEY if in random mode, or a call to either if in dynamic mode. CKDELETE can be called only if the file is currently open for both input and output (input/output type =2). This allows the record to be read into your program's data area and then written back to the file with the delete mark. Following execution of CKDELETE, the deleted record can no longer be accessed.

If the file was opened for shared access with CKOPENSHR, you must lock the file with CKLOCK before you can delete any records with CKDELETE. Because CKDELETE depends on the logical record pointer, the call to CKLOCK should precede the call that positions the pointer. The call to CKUNLOCK is then called after the call to CKDELETE. To illustrate, the sequence of calls in shared access should be:

```
  CKLOCK <--- to lock file
  CKSTART or CKREADBYKEY <--- to position pointer
⋮
  CKDELETE<--- to delete record at which pointer is positioned
  CKUNLOCK<--- to unlock file
```

Following the call to CKDELETE, the pointer is positioned to the next key following the key in the deleted record.

The following examples show the use of CKDELETE for sequential access using CKREAD and for random access using CKREADBYKEY. The WORKING-STORAGE SECTION from Figure A-2 and the FINISH procedure from the CKCLOSE example are assumed for these examples.

Note

If access is shared, the file must be opened with a call to CKOPENSHR and then locked before the call to CKSTART that initially sets the pointer. The file must remain locked while the records to be deleted are read and then marked for deletion. If the file is not locked before CKSTART is called, other users can change the file so that the record pointer points to the wrong record.

In the first example, to delete all records whose primary key begins with "P", first position the file to the start of these records with CKSTART and then read each record with CKREAD and delete it with CKDELETE.

```
WORKING-STORAGE SECTION.
77 RELOP PIC S9(4) COMP.
77 KEYVAL PIC X(20).
77 KEYLOC PIC S9(4) COMP.
77 KEYLENGTH PIC S9(4) COMP.
⋮
PROCEDURE DIVISION.

START.

 MOVE 2 TO I-O-TYPE.
 MOVE 0 TO A-MODE.
 CALL "CKOPEN" USING FILETABLE, STAT.
⋮
```

```
 FIND-REC.
   MOVE 0 TO RELOP.<---  test for equality between
                              primary key and KEY
   MOVE "P" TO KEYVAL.
   MOVE 3 TO KEYLOC.
   MOVE 1 TO KEYLENGTH.<---  check first character only
   CALL "CKSTART" USING FILETABLE, STAT, RELOP, KEYVAL, KEYLOC,
   KEYLENGTH.
   IF STATUS-KEY-1 = "0" THEN
     GO TO READ-REC.
   IF STAT = "23" THEN
     DISPLAY "NO RECORD FOUND"
     GO TO FINISH.
   IF STATUS-KEY-1 = "9" THEN
     CALL "CKERROR" USING STAT, RESULT
     DISPLAY "CKERROR NO.=", RESULT
     GO TO FINISH.
 READ-REC.
   CALL "CKREAD" USING FILETABLE, STAT, REC, RECSIZE.
   IF STATUS-KEY-1 = "1" THEN
DISPLAY "END OF FILE REACHED"
GO TO FINISH.
   IF STATUS-KEY-1 = "0" THEN
   IF NAME OF REC NOT LESS THAN "Q "THEN
DISPLAY "DELETIONS COMPLETED"
     GO TO FINISH;
   ELSE GO TO DELETE-REC;
   ELSE
DISPLAY "CKREAD ERROR, STATUS =", STAT
   IF STATUS-KEY-1 = "9" THEN
CALL "CKERROR" USING STAT, RESULT
DISPLAY "CKERROR NO.", RESULT.
     GO TO READ-REC.

 DELETE-REC.
   CALL "CKDELETE" USING FILETABLE, STAT.
   IF STATUS-KEY-1 = "0" THEN
     DISPLAY "DELETED"
GO TO READ-REC;
   ELSE
DISPLAY "CKDELETE ERROR, STATUS = ", STAT
IF STATUS-KEY-1 = "9" THEN
  CALL "CKERROR" USING STAT, RESULT
  DISPLAY"CKERROR NO.=", RESULT
       GO TO READ-REC.
```

In the second example, a file containing the primary keys of those records to be deleted from a KSAM file is read into the working storage area DAT. These key values are used by CKREADBYKEY to locate and read the items to be deleted by CKDELETE.

```
PROCEDURE DIVISION.

START.
  MOVE 2 TO I-O-TYPE, A-MODE.
  CALL "CKOPEN" USING FILETABLE, STAT.
⋮
READ-KEY.
  READ DATA-FILE INTO DAT;
    AT END GO TO FINISH.
  CALL "CKREADBYKEY" USING FILETABLE, STAT, REC, NAME OF DAT, KEYLOC,
RECSIZE.
  IF STATUS-KEY-1 = "0" THEN
    GO TO DELETE-RECORD.
  DISPLAY "CKREADBYKEY ERROR, STATUS = ",STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKERROR ", RESULT
    GO TO READ-KEY.
DELETE-RECORD.
  CALL "CKDELETE" USING FILETABLE, STAT.
  IF STATUS-KEY-1 = "0" THEN
    DISPLAY REC, " DELETED"
    GO TO READ-KEY.
  DISPLAY "CKDELETE ERROR, STATUS =",STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKERROR NO. =", RESULT.
  GO TO READ-KEY.
```

Note

If access is shared, the file must be opened with a call to CKOPENSHR. A call to CKLOCK must precede the call to CKREADBYKEY. A call to CKUNLOCK must follow the CKDELETE error tests and should precede the return to READ-KEY.

CKERROR

Converts KSAM file system error code returned in *status* to a display format number.

```
CALL "CKERROR" USING status, result
```

Whenever a 9 is returned as the left character of the status parameter following any call to a KSAM procedure, you can call the procedure `CKERROR` to convert the MPE file system error code in the right character of *status* from a binary number to a display format number. This allows you to display the error code.

Parameters

status — The status parameter to which a value was returned by a previous KSAM procedure call. The entire status parameter, both left and right characters, must be specified.

result — An item to which the error number is returned right justified in display format. The item must have a picture of 4 numeric characters (PIC 9(4)).

Operation Notes

The following example shows the WORKING-STORAGE SECTION entries needed to check for errors and a call to `CKERROR` in the PROCEDURE DIVISION that checks for and displays the error number if a file system error occurred in a call to process a KSAM file.

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.
77 RESULT PIC 9(4) VALUE ZERO.
01 STAT.
   03 STATUS-KEY-1 PIC X.
   03 STATUS-KEY-2 PIC X.
.
.
.
PROCEDURE DIVISION.
START.
.
.
.
   IF STATUS-KEY-1 = "9" THEN
      CALL "CKERROR" USING STAT, RESULT.
      DISPLAY "ERROR NUMBER ",RESULT.
```

CKLOCK

A call to CKLOCK dynamically locks a KSAM file.

```
CALL "CKLOCK" USING filetable, status, lockcond
```

When access is shared, you must lock the file before calling CKWRITE, CKREWRITE, or CKDELETE. This ensures that another user cannot attempt to modify the file at the same time. It guarantees that the most recent data is available to each user who accesses the file.

In order to call CKLOCK, the file must have been opened with a call to CKOPENSHR, not CKOPEN.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so, what it was. (Refer to Filetable Parameter discussion earlier in this section.)

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKLOCK. It indicates whether or not the file was successfully locked and if not, why not. The *status* word = 00 if the call was successful. It = 30 if the file was locked by another process. It = 9n, where n is a file system error code, if the call failed for some other reason. (Refer to the Status Parameter discussion earlier in this section.)

lockcond — One halfword computational item whose value determines the action taken if the file is locked by another user when CKLOCK is executed. The value is either zero (0) or one (1).

- 0 locking is conditional; if the file is already locked, control is returned to your program immediately with the *status* word set to "30".
- 1 locking is unconditional; if the file cannot be locked immediately because another use has locked it, your program suspends until the file can be locked.

Operation Notes

In order to call CKLOCK, the file must be opened with dynamic access enabled. This can be done only with the CKOPENSHR procedure. CKOPEN will not open the file for shared access with dynamic locking.

When users are sharing a file, it is essential to lock the file before modifying it. An error is returned if any user attempts to write, rewrite, or delete records without first locking the file. It is also important to avoid situations where one user locks the file and forgets to unlock it. If the file is already locked when you call CKLOCK with *lockcond* set to zero, the call will fail with 30 returned to *status*, and

your process will continue. If, however, *lockcond* is set to 1, your process suspends until the other user unlocks the file or logs off.

The following example opens file KSAMFILE for shared access with dynamic locking allowed. It then locks the file unconditionally. If another user has locked the file, the process suspends until the file is unlocked and then continues by locking your file. The status value is checked as soon as control returns to your process to ensure that the file has been locked before continuing.

```
 DATA DIVISION.

 77   LOCKCOND      PICTURE S9(4)   COMP   VALUE 1.
 77   RESULT        PICTURE 9(4)           VALUE 0.
 01   STATUSKEY.
    02 STATUS-KEY1  PICTURE X VALUE " ".
    02 STATUS-KEY2  PICTURE X VALUE " ".
 01   FILETABLE.
    02  FILENUMBER  PICTURE S9(4)   COMP   VALUE 0.
    02  FILENAME    PICTURE X(8)           VALUE "KSAMFILE".
    02  I-O-TYPE    PICTURE S9(4)   COMP   VALUE 0.
    02  A-MODE      PICTURE S9(4)   COMP   VALUE 0.
    02  PREV-OP     PICTURE S9(4)   COMP   VALUE 0.

 PROCEDURE DIVISION.

START.
    CALL "CKOPENSHR" USING FILETABLE, STATUSKEY.
    IF STATUS-KEY1 = "0" THEN GO TO LOCK-FILE.
    IF STATUS-KEY1 = "9" THEN
  CALL "CKERROR" USING STATUSKEY, RESULT
  DISPLAY "ERROR NO. ",RESULT.

LOCK-FILE.
    CALL "CKLOCK" USING FILETABLE, STATUSKEY, LOCKCOND.
    IF STATUSKEY="00"
       THEN DISPLAY "CKLOCK IS OK"
    ELSE IF STATUSKEY = "30"
       THEN DISPLAY "FILE LOCKED BY ANOTHER PROCESS"
    ELSE IF STATUS-KEY1="9"
       THEN CALL "CKERROR" USING STATUSKEY, RESULT
    DISPLAY "ERROR NO.", RESULT.
```

CKOPEN

A call to procedure CKOPEN initiates KSAM file processing.

```
CALL "CKOPEN" USING filetable, status
```

In order to process a KSAM file, it must be opened with a call to the CKOPEN procedure. CKOPEN initiates processing, specifies the type of processing and the access mode; the file must have been created previously.

To open a file means to make it available for processing, to specify the type of processing (input only, output only, or both), and to specify the access method (sequential, random, or dynamic). If a different type of processing or access method is needed, the file must be closed and opened again with the parameters set to new values.

Note

If you want to open the file for shared access, you must use a call to CKOPENSHR, rather than CKOPEN.

Parameters

filetable — An 8 halfword record containing the name of the file, its input/output type, and access mode. When the open is successful, the first word of this table is set to the file number that identifies the opened file. (Refer to Filetable Parameter discussion earlier in this section.)

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKOPEN to indicate whether or not the file was successfully opened and if not why not. The left character is set to 0 if open is successful, to 9 if not. The right character is set to 0 if the open is successful, to the file system error code if not. (Refer to Status Parameter discussion earlier in this section.)

Operation Notes

Upon successful execution of CKOPEN, the file named in *filetable* is available for the type of processing specified in *filetable*. Before the file is successfully opened with CKOPEN, no operation can be executed that references the file either explicitly or implicitly.

The input/output procedures that can be called to process the file depend on the value of the halfwords in *filetable* that specify input/output type and access mode. (Refer to Figure A-3 for the procedures allowed with the various combinations of input/output type and access mode.)

A file may be opened for input, output, or input/output, and for sequential, random, or dynamic access in the same program by specifying a different call to CKOPEN for each change in input-output type or access mode. Following the initial execution of CKOPEN, each subsequent call to CKOPEN for the same file must be preceded by a call to CKCLOSE for that file.

When files are opened for input or input/output, the call to CKOPEN sets the current record pointer to the first record in the primary key chain.

ALLOWED PROCEDURES	ACCESS MODE		INPUT-OUTPUT TYPE
CKREAD CKSTART	0 (sequential)	2 (dynamic)	0 (open for input)
CKREADBYKEY	1 (random)		
CKWRITE	0 (sequential)	2 (dynamic)	1 (open for output)
	1 (random)		
CKREAD CKSTART CKREWRITE CKDELETE	0 (sequential)	2 (dynamic)	2 (open for input/output)
CKREADBYKEY CKWRITE CKREWRITE CKDELETE	1 (random)		

Figure A-3. Procedures Allowed for Input/Output Type/Access Mode Combinations

Halfword 6 of *filetable* must be set to one of the following values before calling CKOPEN:

0 input only

1 output only

2 input/output

In general, if you want to allow records to be read or the file to be positioned without allowing any new records to be written or any existing records to be changed, you should set the input/output type to 0. This input/output type allows you to call CKREAD or CKSTART in sequential processing mode, CKREADBYKEY in random mode, or all three in dynamic mode.

If you want to cause all existing records to be deleted when the file is opened and then allow new records to be written, you should set the input/output type to 1. This type of open deletes all existing records so that records are written to an empty file. When a file is opened for output only, you can call CKWRITE in any of the three access modes: sequential, random, or dynamic, but you cannot call any other of the KSAM procedures.

If you want unrestricted file access, you should set the input/output type to 2. This access type allows records to be read, positioned, written, rewritten, or deleted. You may call CKREAD, CKSTART, CKREWRITE, and CKDELETE (but not CKWRITE) when opened in sequential mode; you may call CKREADBYKEY, CKWRITE, CKREWRITE, or CKDELETE (but not CKREAD or CKSTART) when opened in random mode. In dynamic mode, any of the KSAM procedures may be called. With this type of input/output, existing records are not cleared when you write a record with CKWRITE.

Halfword 7 of *filetable* must be set to one of the following values before calling CKOPEN:

0 sequential access

1 random access

2 dynamic access

With sequential access, records in the file are read in ascending order based on the value of a key within each record. The key is the primary key unless an alternate key was specified with CKSTART. Reading starts with the first record in sequence unless a particular record was specified with CKSTART. Each time a call to CKREAD is executed, the next record in sequence is read from the file. CKREAD and CKSTART are the only procedures that can be called in input mode. CKREADBYKEY cannot be specified for any input/output type if the access mode is sequential.

In output mode, CKWRITE is the only procedure that can be called. When access is sequential, the record to be written must contain a unique primary key that is greater in value than the key of any previously written record. If it is not in sequence, an invalid key sequence error 21 is returned to *status*.

In input/output mode, CKREWRITE and CKDELETE can be specified as well as CKREAD and CKSTART, but CKWRITE cannot.

Random access allows you to read, write, replace, or delete a record with any value for its primary key. To read a record, the CKREADBYKEY procedure must be called in either input or input/output mode. CKREAD and CKSTART cannot be specified for any input/output type when access mode is random.

When writing a record with CKWRITE in output or input/output mode, the value of the primary key in the record need not be greater than the keys of previously written records; that is, records can be written in any order.

In input/output mode, CKREWRITE can be used to replace any record whose primary key matches the primary key in the record being written. CKDELETE can be used to delete a record specified in a previous CKREADBYKEY call.

CKWRITE can be used to write a record following existing records in the file if you position to follow the last sequential record before

writing. Use this input/output type if you want to save existing data in a file to which you are writing.

Dynamic access allows you to use any call to process a file opened for input/output. When the file is opened in dynamic mode, and a call is made to CKREAD or CKSTART, the file can be read, but not updated, sequentially. For all other calls, dynamic mode is treated as if the file had been opened in random mode. The reason to open a file in dynamic mode is to allow both sequential and random processing on the same file without closing it and then opening it again each time access switches from sequential to random or vice versa.

To open a file initially for sequential read:

```
   WORKING-STORAGE SECTION.
   77 RESULT PIC 9(4) VALUE ZERO.
   01 FILETABLE.
      03 FILENUMBER PIC S9(4) COMP VALUE ZERO.
      03 FILENAME PIC X(8) VALUE "KSAMFILE".
      03 I-O-TYPE PIC S9(4) COMP VALUE ZERO.<---  input only
      03 A-MODE PIC S9(4) COMP VALUE ZERO.<-----  sequential access
      03 PREV-OP PIC S9(4) COMP VALUE ZERO.
   01 STAT.
      03 STATUS-KEY-1 PIC X.
      03 STATUS-KEY-2 PIC X.
    .
    .
    .
   PROCEDURE DIVISION.

   START.
      CALL "CKOPEN" USING FILETABLE, STAT.
      IF STATUS-KEY-1 ="0" THEN GO TO S-READ.
      IF STATUS-KEY-1 ="9" THEN
        CALL "CKERROR" USING STAT, RESULT
      DISPLAY "CKOPEN FAILED. . .ERROR NO.", RESULT
      STOP RUN.
 S-READ.
      .
      .
      .
```

If you subsequently want to write in sequential order to the same file, you should close the file with a call to CKCLOSE (described below), move the value 1 (output to I-O-TYPE and then reopen the file:

```
CALL "CKCLOSE" USING FILETABLE, STAT.
IF STATUS-KEY-1 ="9" THEN
  CALL "CKERROR" USING STAT, RESULT
  DISPLAY "CKCLOSE FAILED -- ERROR NO.",
  STOP RUN.
MOVE 1 TO I-O-TYPE.<--- output only
CALL "CKOPEN" USlNG FILETABLE, STAT.
```

Similarly, to update records in random order in the same file, first close the file, then use the following `MOVE` statement to alter the input/output type and access mode in `FILETABLE` and reopen the file:

```
CALL "CKCLOSE" USING FILETABLE, STAT.
⋮
MOVE 2 TO I-O-TYPE.<--- input/output
MOVE 1 TO A-MODE.<--- random access
CALL "CKOPEN" USING FILETABLE, STAT.
```

CKOPENSHR

A call to CKOPENSHR initiates KSAM file processing with dynamic locking and shared access allowed.

```
CALL "CKOPENSHR" USING filetable, status
```

In order to process a KSAM file with shared access and dynamic locking, the file must be opened with a call to CKOPENSHR. CKOPENSHR is exactly like CKOPEN in that it initiates processing, specifies the type of processing, and specifies the access mode. The file must have been created previously.

To open a file for shared access means to make it available for processing by more than one user. Shared access allows all users to read or position the file, but only one user at a time can modify the file by writing new records, or rewriting or deleting existing records. To ensure that more than one user does not attempt to modify the file at the same time, you must call CKLOCK to dynamically lock the file before calling the procedures CKWRITE, CKREWRITE, or CKDELETE. After modifying the file, you should call CKUNLOCK so that it can be accessed by other users.

Parameters

filetable — An 8 halfword record containing the name of the file, its input/output type, and access mode. When the open is successful, the first halfword of this table is set to the file number that identifies the opened file.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKOPENSHR to indicate whether or not the file was successfully opened and if not why not. The left character is set to 0 if the open is successful, to 9 if not. The right character is to 0 if open is successful, to the file system error code if not.

Operation Notes

A call to CKOPENSHR operates like the call to CKOPEN, except that CKOPENSHR allows shared access and dynamic locking. Upon successful execution of CKOPENSHR, the file named in *filetable* is available for the type of processing specified in *filetable*. Before the file is opened successfully, no operation can be performed that references the file either explicitly or implicitly.

A file may be opened by CKOPENSHR for any of the access modes (sequential, random, or dynamic) and for any input/output type (input only, output only, or input/output) allowed with CKOPEN.

Refer to the description of using CKOPEN for the specific effects of opening a KSAM file with the various input/output types and access modes.

CKREAD

A call to procedure CKREAD makes available the next logical record from a KSAM file.

```
CALL "CKREAD" USING filetable, status, record, recordsize
```

In order to read records in sequential order by key value, call procedure CKREAD. The file must have been opened in input or input/output mode with access mode specified as either sequential or dynamic.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so, what it was.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKREAD to indicate whether or not the record was successfully read and if not, why not.

record — A record defined in the WORKING-STORAGE SECTION into which the contents of the next sequential KSAM record is read.

recordsize — An integer (S9(4)COMP) containing the length in characters of the record being read. It must not exceed the maximum record length established for the file when it was created.

Operation Notes

The file from which the record is read must be opened for sequential or dynamic access (access mode = 0 or 2). It may be opened for input only or input/output (input/output type = 0 or 2), but not for output only.

When the file is opened initially for input or input/output, the logical record pointer is positioned at the first sequential record; that is, at the record with the lowest key value. The key used is the primary key unless a previous call to CKSTART has specified an alternate key. When a call to CKREAD is executed, the record at which the record pointer is currently positioned is read into the location specified by *record*.

If, when CKREAD is executed, there is no next logical record in the file, the at end condition is returned to *status*; that is, *status* is set to 10. Note that a call to the procedure CKSTART can be used to reposition the pointer for subsequent sequential access according to primary or alternate key order.

In order to update records in sequential order, CKREAD must be called before executing either of the update procedures CKREWRITE or CKDELETE. When access is shared, it is important to include the call to CKREAD within the same locked portion of code that includes the

call to CKREWRITE or CKDELETE. This ensures that the correct record is modified or deleted.

Because CKREAD is a pointer-dependent procedure, the actual record read depends on the current position of the logical record pointer. When access is shared, this pointer position can be made incorrect by other users without your program being aware of it. For this reason, you should lock the file, position the pointer with a pointer-independent procedure, and then call CKREAD. When the last record is read, you should then unlock the file so other users can access the file. Example 2 below illustrates how you should read the file sequentially when access is shared.

Using the WORKING-STORAGE SECTION from Figure A-2 and the FINISH procedure in the CKCLOSE example, the following procedures read records in sequential order from file KSAMFILE and display them on the standard output device.

```
PROCEDURE DIVISION.

START.
  ⋮
  MOVE 0 TO I-O-TYPE, A-MODE.
  CALL "CKOPEN" USING FILETABLE, STAT.
  IF STATUS-KEY-1 = "9"
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKOPEN ERROR NO. ", RESULT.
  IF STATUS-KEY-1 NOT = "0"
    DISPLAY "CKOPEN FAILED"
  STOP RUN.
    READ-NEXT.
      CALL "CKREAD" USING FILETABLE, STAT, REC, RECSIZE.
      IF STATUS-KEY-1 = "1" GO TO NEW-POSITION.
      IF STATUS-KEY-1 = "0"
        DISPLAY REC;
      ELSE
        DISPLAY "CKREAD ERROR, STATUS =", STAT.
      IF STATUS-KEY-1 ="9"
        CALL "CKERROR" USING STAT, RESULT
        DISPLAY "FILE ERROR =", RESULT.
      GO TO READ-NEXT.
  NEW-POSITION.
    ⋮
```

The following example provides a sequential read with shared access.

```
PROCEDURE DIVISION.
START.
⋮
 MOVE 0 TO I-O-TYPE, A-MODE.
 CALL "CKOPENSHR" USING FILETABLE, STAT <--- open file for shared
access
⋮ <--- test status
FIND-RECORD.
 MOVE 2 TO RELOP.
 MOVE "000-0000" TO KEYVAL.
 MOVE 23 TO KEYLOC,
 MOVE 8 TO KEYLENGTH.
 MOVE 1 TO LOCKCOND.
 CALL "CKLOCK" USING FILETABLE, STAT, LOCKCOND.<--- lock file
unconditionally
 CALL "CKSTART" USING FILETABLE,
   STAT, RELOP, KEYVAL, KEYLOC, KEYLENGTH.<--- position pointer to
lowest key value
⋮ <--- test status
READ-RECORD.
 CALL "CKREAD" USING FILETABLE, STAT, REC, RECSIZE<--- read record
 IF STATUS-KEY-1 ="1"<--- end of file
   GO TO END-OF-READ.
 IF STATUS-KEY-1 ="0"<--- if successful, display record read
   DISPLAY REC.
⋮ <--- test status for errors
 TO TO READ-RECORD.
END-OF-READ.
 CALL "CKUNLOCK" USING FILETABLE, STAT.<----- unlock file
```

CKREADBYKEY

A call to CKREADBYKEY makes available a record identified by key value from a KSAM file.

```
CALL "CKREADBYKEY" USING filetable, status, record, key, keyloc, recordsize
```

Records can be read from a KSAM file in an order determined by key value. This order need not be sequential; in fact, it can be any order you specify. This type of access is used to access individual records in random order by key value.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so what it was.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKREADBYKEY indicating whether the call was successful and if not why not.

record — A record defined in the WORKING-STORAGE SECTION into which the contents of a record located by key value is read.

key — An item whose value is used by CKREADBYKEY to locate the record to be read. Key values in the file identified by *filetable* are compared to the value of *key* until the first record with an equal value is found.

keyloc — One halfword integer (S9(4)COMP) set to the starting character position of the key in the KSAM data record (first position is character 1). The *keyloc* parameter identifies the file key to be compared with *key*.

recordsize — An integer (S9(4)COMP) containing the length in characters of the record being read; it must be less than or equal to the maximum record length established for the file at creation.

Operation Notes

In order to use the CKREADBYKEY procedure, the file must be opened for either input or input/output. The access mode can be either random or dynamic, but must not be sequential.

Execution of CKREADBYKEY causes the value of *key* to be compared to the value of the key at location *keyloc* in the KSAM file data records. When a key is found whose value is identical to that of *key*, the record pointer is moved to the beginning of that record and the record is read into the location *record*.

If no record can be found whose key value equals that of *key*, an invalid key condition is diagnosed and *status* is set to the value 23.

Successful execution of CKREADBYKEY is indicated by the value 0 in the left byte of *status*. Unsuccessful execution is indicated by either the invalid key return or by a value of 9 in the left byte of *status*.

In order to delete records in random or dynamic mode, CKREADBYKEY must be called before executing CKDELETE. It is not required prior to CKREWRITE.

In the following examples, update information is read into the area called DAT in the WORKING-STORAGE SECTION. (Note that in this as in the preceding examples, the WORKING- STORAGE SECTION from Figure A-2 continues to be useful.) In the first example, the primary keys of records in KSAMFILE are searched for values matching the value read into *NAME* in the DAT record; in the second example, an alternate key at location 23 is searched for values matching the value read into *PHONE* in the DAT record.

Read a record located by its primary key value:

```
DATA DIVISION.
⋮
WORKING-STORAGE SECTION.
77 KEYLOC PIC S9(4) COMP.
⋮
PROCEDURE DIVISION.
START.
⋮
  MOVE 2 TO I-O-TYPE, A-MODE.<--- prepare to open for input/output, dynamic access
  CALL "CKOPEN" USING FILETABLE, STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKOPEN ERROR NO. ", RESULT.
  IF STATUS-KEY-1 NOT="0" THEN
    DISPLAY "CKOPEN FAILED"
    STOP RUN.
FIND-RECORD.
  READ NEW-DATA INTO DAT;<--- read update records
    AT END GO TO FINISH.
  MOVE 3 TO KEYLOC.
  CALL "CKREADBYKEY" USING FILETABLE, STAT, REC, NAME OF DAT,
    KEYLOC, RECSIZE.
  IF STAT = "00" THEN
    DISPLAY "RECORD FOUND", REC
    GO TO FIND-RECORD.
  IF STAT = "23" THEN
    DISPLAY "RECORD NOT FOUND,KEY=", NAME OF DAT
    GO TO FIND-RECORD.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "ERROR NO. ", RESULT
  GO TO FIND-RECORD.
```

To find a record by the value of an alternate key, simply change two statements in the preceding example so that *KEYLOC* contains the location of the alternate key and the *key* value for comparison is found in item *PHONE OF DAT* rather than in *NAME OF DAT*:

```
FIND RECORD.
  READ NEW-DATA INTO DAT;
    AT END GO TO FINISH.
  MOVE 23 TO KEYLOC.
  CALL "CKREADBYKEY" USING FILETABLE, STAT, REC, PHONE OF DAT,
    KEYLOC, RECSIZE.
```

CKREWRITE

The procedure CKREWRITE replaces a record existing in a KSAM file with another record having a matching primary key.

```
CALL "CKREWRITE" USING filetable, status, record, recordsize
```

You can replace an existing record in a KSAM file with the procedure CKREWRITE. This procedure replaces a record previously read from the file with another record whose primary key matches the primary key of the record being replaced.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was unsuccessful and if so what it was.

status — One halfword (two 8-bit characters) set to a pair of values upon the completion of the call to CKREWRITE indicating whether or not the call was successful and if not why not. (Refer to Status Parameter discussion earlier in this section.)

record — A record defined in the WORKING-STORAGE SECTION containing data to be written as a logical record to the file replacing the record with a matching primary key.

recordsize — An integer (S9(4)COMP) containing the length in characters of the record to be written. It must not exceed the maximum record length established for the file when it was created.

Operation Notes

In order to call procedure CKREWRITE, the file must be open for both input and output (input/output type=2). The access mode can be sequential, random, or dynamic. If access mode is sequential, CKREAD must have been executed successfully just prior to the call to CKREWRITE. In random or dynamic mode, no prior read is required; the system searches the file for the record to be rewritten.

When the file is opened in sequential mode (access mode = 0), CKREAD must be executed before CKREWRITE. The primary key in the record to be written by CKREWRITE must be identical to the primary key in the record read by CKREAD. A simple way to ensure that the keys match is to read a record into WORKING-STORAGE, modify it without altering the primary key, and then write it back to the file using CKREWRITE. Since the primary key is not changed, the sequence of records in the file is not affected.

If you want to rewrite in sequential mode all the records in a chain of records with duplicate keys, use either CKSTART or CKREADBYKEY to position to the first record in the chain. Then call CKREWRITE to update the first record in the chain. Subsequent calls depend on whether you are changing any key value in the record (not necessarily the selected key).

If no key in the record is changed, the record pointer continues to point to the current record. Only a subsequent CKREAD advances the pointer to the next record in the duplicate key chain. In this case, you can issue CKREAD and CKREWRITE calls until all records with the duplicated key value have been rewritten.

If any key in the record is changed, the new key is written to the end of the chain of duplicate keys in the index area. After the first call to CKREWRITE, the record pointer points to the record whose key value follows the changed key. Since this key is now at the end of the chain of duplicate keys, a subsequent call to CKREWRITE skips all records with keys in the duplicate key chain and rewrites the record with the next higher key value. In this case, you must precede each call to CKREWRITE with a call to CKSTART or CKREADBYKEY in order to update all subsequent records with duplicate keys.

If you are updating a primary key value that is duplicated, it is good practice to use CKDELETE to delete the selected record and then rewrite it as a new record with CKWRITE.

When the file is opened in random or dynamic mode (access mode = 1 or 2), no prior call to a read procedure is needed. You specify the record to be written in WORKING-STORAGE and then call CKREWRITE. However, you must use the primary key to position to the record to be modified. When the procedure is executed, the file is searched for a record whose primary key matches that of the record to be written. If such a record is found, it is replaced by the record specified in CKREWRITE. If not found, an invalid key condition is diagnosed and *status* is set to 23.

A call to CKREWRITE in random mode updates only the first record with a key in the chain of duplicate keys.

Regardless of the mode, after any call to CKREWRITE that does not modify a key value, the record pointer is positioned to the key of the record just modified. However, if any key in the modified record was changed, the record must be deleted and then rewritten by a write procedure. If the access mode is sequential and a key was modified, the pointer is moved to the record with the next key value in ascending sequence after the modified key. If the access mode is random or dynamic, and a key was modified, the pointer is moved to the record with the next key in ascending sequence after the *primary* key in the modified record. This means that in random or dynamic mode the key pointer may change if it was pointing to an alternate key before the call to CKREWRITE.

If the file was opened for shared access with CKOPENSHR, then you must lock the file with a call to CKLOCK before rewriting any records with CKREWRITE. After the records are rewritten, you should unlock the file with CKUNLOCK.

To ensure that you are updating the correct record in sequential mode, you should call CKLOCK before positioning the pointer with CKSTART or CKREADBYKEY, then specify the sequential calls to CKREAD

and CKREWRITE before unlocking the file with CKUNLOCK. This ensures that no other users change the position of the pointer while you are sequentially updating the file.

In sequential mode, the invalid key condition exists when the record just read by CKREAD and the record to be written by CKREWRITE do not have the same primary key value. In random or dynamic mode, an invalid key condition exists if no record can be found in the file whose primary key matches that of the record to be written by CKREWRITE. In either case, *status* is set to the value 23.

Regardless of mode, an invalid key condition occurs if an alternate key value in the record to be written duplicates a corresponding alternate key for which duplicates are prohibited. When rewriting a record, try to avoid specifying an alternate key value that may duplicate a value existing in the file unless duplicates are allowed for the key. A duplicate key condition where duplicates are not allowed causes *status* to be set to 22 and the procedure is not executed.

Use CKSTART to position the current record pointer to the start of the file. Then read each record in sequence and set its non-key items to blanks.

The first example is of a sequential update that clears the value of an item in each record of the file. The second example searches the file for a record whose primary key has a particular value in order to change the alternate key for that record. Both examples assume the WORKING-STORAGE SECTION from Figure A-2 and the FINISH procedure from CKCLOSE.

Note

If the file was opened for shared access with a call to CKOPENSHR, then the file should be locked with a call to CKLOCK before the call to CKSTART. The file should be unlocked with a call to CKUNLOCK only when the final record is updated, probably in the FINISH procedure.

```
DATA DIVISION.
⋮
WORKING-STORAGE SECTION.        \
77 RELOP        PIC S9(4)   COMP.|
77 KEYVAL       PIC X(20).       |<---  items required by CKSTART
77 KEYLOC       PIC S9(4)   COMP.|
77 KEYLENGTH    PIC S9(4)   COMP.|
⋮
PROCEDURE DIVISION.
START.
  MOVE 2 TO I-O-TYPE.
  MOVE 0 TO A-MODE.
  CALL "CKOPEN" USING FILETABLE, STAT.
⋮ <---  check status
UPDATE-FILE.
  MOVE 1 TO RELOP.
  MOVE "000-0000" TO KEYVAL.<---  set up CKSTART parameters to start
  MOVE 23 TO KEYLOC.               reading at lowest alternate key
value
  MOVE 8 TO KEYLENGTH.
  CALL "CKSTART" USING FILETABLE, STAT, RELOP, KEYVAL, KEYLOC,
KEYLENGTH.
  IF STATUS-KEY-1="0" THEN
    GO TO READ-RECORD;
  ELSE
    DISPLAY "CKSTART ERROR, STATUS", STAT.
    IF STATUS-KEY-1 = "9" THEN
      CALL "CKERROR" USING STAT, RESULT
      DISPLAY "CKERROR NO.", RESULT
    GO TO FINISH.
READ-RECORD.
  CALL "CKREAD" USING FILETABLE, STAT, REC, RECSIZE.
  IF STATUS-KEY-1 = "1" THEN
    GO TO FINISH. <------------------  end of file
  IF STATUS-KEY-1 = "0" THEN
    GO TO WRITE-RECORD
  ELSE
    DISPLAY "CKREAD ERROR,STATUS =", STAT.
    IF STATUS-KEY-1 = "9" THEN
      CALL "CKERROR" USING STAT, RESULT
      DISPLAY "CKERROR NO. ", RESULT
    GO TO READ-RECORD.
```

```
WRITE-RECORD.
  MOVE SPACES TO OTHERDATA OF REC.
  CALL "CKREWRITE" USING FILETABLE,
  IF STATUS-KEY-1 = "0" THEN
    DISPLAY NAME OF"DATA CLEARED"
    GO TO READ-RECORD.
  DISPLAY "CKREWRITE ERROR, STATUS=",
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT,
    DISPLAY "CKERROR NO.=",
    GO TO READ-RECORD.
```

The second example finds the record with the primary key "ECKSTEIN, LEO "and changes the value of the secondary key to "257-5137":

```
PROCEDURE DIVISION.

START.
⋮
  MOVE 2 TO I-O-TYPE, A-MODE.
  CALL "CKOPEN" USING FILETABLE, STAT.
  IF STATUS-KEY-1 = "0" THEN
    GO TO F-UPDATE.
  DISPLAY "CKOPEN ERROR, STA", STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKERROR NO.=", RESULT
  GO TO FINISH.
F-UPDATE.
  MOVE "ECKSTEIN, LEO "TO NAME OF REC.
  MOVE "257-5137" TO PHONE OF REC.
  MOVE SPACES TO OTHERDATA OF REC.
  CALL "CKREWRITE" USING FILETABLE, STAT, REC, RECSIZE.
  IF STATUS-KEY-1="0" THEN
    DISPLAY REC "UPDATED"
    GO TO FINISH.
  IF STAT = "23" THEN
    DISPLAY NAME OF REC "NOT FOUND"
    GO TO FINISH.
  DISPLAY "CKREWRITE ERROR, STATUS =", STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "CKERROR NO.=", RESULT.
  GO TO FINISH.
```

CKSTART

A call to procedure CKSTART allows you to position the record pointer to a particular record in a KSAM file defined by its primary or alternate key value.

```
CALL "CKSTART" USING filetable, status, relop, key, keyloc, keylength
```

In order to position the current record pointer to a location in the file defined by a key value, call CKSTART. Since CKSTART is used in preparation for sequential retrieval of records with CKREAD, the file must be open for sequential or dynamic access, not random, and for input or input/output, not output only.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so, what it was.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKSTART to indicate whether or not the call was successful and if not why not. (Refer to Status Parameter discussion earlier in this section.)

relop — One halfword integer (S9(4)COMP) code that specifies a relation between the key value specified in the call to CKSTART and the key value in the record to which the record pointer is to be positioned:

0—record key is equal to *key*
1—record key is greater than *key*
2—record key is greater than or equal to *key*

key — An item whose value is used by CKSTART to locate the record at which to position the record pointer. The values of a specified file key are compared in ascending order to the value of *key* according to the relation specified by *relop*.

keyloc — One halfword integer (S9(4)COMP) set to the starting character location of a key in the KSAM file data record (first position is character 1). The key at *keyloc* is compared to *key*.

keylength — One halfword integer (S9(4)COMP) set to the length of *key*; the length must be less than or equal to the length of the key defined by *keyloc*.

Operation Notes

When CKSTART is executed, the index area is searched for the first key in the set of keys at location *keyloc* whose value when compared with *key* satisfies the comparison specified by *relop*. The current record pointer is positioned to the beginning of the record in the data area associated with the key found by CKSTART.

The specified length of *key* (*key length*) may be less than the length of the key in the file; if so, the comparison proceeds as if the file key were truncated on the right to the same length as *key length*. If no record can be found whose key value satisfies the comparison, an invalid key condition is returned to *status*; that is, *status* is set to 23.

If you use CKSTART to position the pointer before reading or updating the file sequentially in a shared environment, you must lock the file with a call to CKLOCK before calling CKSTART. Then, after you have completed the sequential operations, you can unlock the file with a call to CKUNLOCK. If you wait to lock the file until after the call to CKSTART, another user can change the structure of the index area so that the position of the pointer becomes invalid for any subsequent call to a procedure that depends on the pointer position.

For the following examples, four new items must be added to the WORKING-STORAGE SECTION in Figure A-2; otherwise, the same WORKING-STORAGE SECTION is used. The new items are:

```
77   RELOP       PIC S9(4)  COMP.
77   KEYVAL      PIC X(20).
77   KEYLOC      PIC S9(4)  COMP.
77   KEYLENGTH   PIC S9(4)  COMP.
```

Each of these items is assigned the value appropriate to the operation to be performed by statements in the PROCEDURE DIVISION. Note that the length of array *KEYVAL* can be made shorter by assigning a value less than 20 to *KEYLENGTH* but it cannot be made longer than 20 characters. Since there is no key in KSAMFILE longer than 20 characters, this allows comparison to be made on the longest key.

The following example shows the statements needed to display the records in KSAMFILE in order by the alternate key *PHONE* that starts in location 23 and has a length of 8 characters. It assumes the file is open for input or input/output and that the access mode is sequential. It also assumes the FINISH procedure from the CKCLOSE example.

```
NEW-POSITION.
  MOVE 2 TO RELOP.<--- find key value greater than or equal to
KEYVAL
  MOVE "000-0000" TO KEYVAL.
  MOVE 23 TO KEYLOC.
  MOVE 8 TO KEYLENGTH.
  CALL "CKSTART" USING FILETABLE, STAT, RELOP, KEYVAL, KEYLOC,
KEYLENGTH.
  IF STAT = "23" THEN GO TO FINISH.<--- no record found
  IF STATUS-KEY-1 = "0" THEN GO TO READ-BY-PHONE.<--- lowest key
value found
  DISPLAY "CKSTART ERROR, STATUS", STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "ERROR NUM", RESULT.
  GO TO FINISH.

READ-BY-PHONE.
  CALL "CKREAD" USING FILETABLE, STAT, REC, RECSIZE,
  IF STATUS-KEY-1 = "1" THEN GO TO FINISH.<---- end-of-file
  IF STATUS-KEY-1 = "0" THEN
    DISPLAY REC;
  ELSE DISPLAY "CKREAD ERROR,STATUS=", STAT
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "ERROR NUMBER", RESULT.
  GO TO READ-BY-PHONE.
```

In the next example, CKSTART is used to position to the beginning of the series of names beginning with the letter "T". The KSAM file key is located at character position 3 (*NAME* key); the parameter *KEYVAL* is set to the value "T"; the key length for purposes of comparison is set to 1; and *RELOP* is set to 0. Thus the record pointer is positioned at the first key found whose value (when the key is truncated to 1 character) is equal to "T". Note that this example reads not only all names beginning with "T", but also reads all names that begin with letters following "T". To read only the names beginning with "T", the program must add a test for the end of the "T" names.

```
POSITION.
  MOVE 0 TO RELOP.<--- find key equal to KEY value
  MOVE "T" TO KEYVAL.
  MOVE 3 TO KEYLOC.
  MOVE 1 TO KEYLENGTH.
  CALL "CKSTART" USING FILETABLE, STAT, RELOP, KEYVAL, KEYLOC,
KEYLENGTH.
  IF STAT = "23" THEN GO TO FINISH.
  IF STATUS-KEY-1 = "0" THEN
    GO TO READ-NAMES.
  DISPLAY "CKSTART ERROR, STATUS=",STAT.
  IF STATUS-KEY-1 = "9" THEN
    CALL "CKERROR" USING STAT, RESULT
    DISPLAY "ERROR NUMBER=", RESULT.
  GO TO FINISH.
READ-NAMES.
  CALL "CKREAD" USING FILETABLE, STAT, REC, RECSlZE.
  IF STATUS-KEY-1 ="1" THEN GO TO FINISH.
  IF STATUS-KEY-1 ="0" THEN
    DISPLAY REC;
  ELSE
    DISPLAY "CKREAD ERROR, STATUS",STAT.
    IF STATUS-KEY-1 = "9" THEN
      CALL "CKERROR" USING STAT, RESULT
      DISPLAY "ERROR NUM", RESULT.
  GO TO READ-NAMES.
```

CKUNLOCK

A call to CKUNLOCK unlocks a KSAM file dynamically locked by CKLOCK.

```
CALL "CKUNLOCK" USING filetable, status
```

A file locked by CKLOCK is released for use by other users with a call to CKUNLOCK. (If you log off from any connection with the system, the file is also unlocked.) Since dynamic locking takes place during shared access to the same file by more than one user, it is important that any file locked by CKLOCK be unlocked as soon as possible by CKUNLOCK.

To use CKUNLOCK, the file must be opened for shared access with dynamic locking allowed. This can be done only by calling CKOPENSHR to open the file, not CKOPEN.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation was successful and if so, what it was.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKUNLOCK. It indicates whether or not the file was successfully unlocked and if not, why not. The *status* word is set to 00 if the file was unlocked successfully; to 31 if the file was not locked; or to 9*n* where *n* is a binary file system error code if the call fails for any other reason.

Operation Notes

After calling CKUNLOCK, you should always check the status parameter to make sure that the procedure was executed successfully. When successful, the file locked by CKLOCK is again made available for access by other users. If the file was not locked by CKLOCK, when CKUNLOCK is called, *status* is set to 31.

The following example unlocks a file previously locked by CKLOCK. (Refer to the CKLOCK example.)

```
DATA DIVISION.
⋮
77  RESULT           PICTURE 9(4)         VALUE 0.
01  STATUSKEY.
    02  STATUS-KEY1  PICTURE X            VALUE " ".
    02  STATUS-KEY2  PICTURE X            VALUE " ".
01  FILETABLE.
    02  FILENUMBER   PICTURE S9(4)  COMP  VALUE 0.
    02  FILENAME     PICTURE X(8)         VALUE "KSAMFILE".
    02  I-O-TYPE     PICTURE S9(4)  COMP  VALUE 0.
    02  A-MODE       PICTURE S9(4)  COMP  VALUE 0.
    02  PREV-OP      PICTURE S9(4)  COMP  VALUE 0.

PROCEDURE DIVISION.
⋮
 CALL "CKUNLOCK" USING FILETABLE, STATUSKEY.
 IF STATUSKEY ="00"
   THEN DISPLAY "CKUNLOCK IS OK"
 ELSE IF STATUSKEY ="31"
   THEN DISPLAY="FILE NOT PREVIOUSLY LOCKED BY THIS PROCESS"
 ELSE IF STATUS-KEY1 ="9"
   THEN CALL"CKERROR" USING STATUSKEY, RESULT
 DISPLAY "ERROR NO.", RESULT.
```

CKWRITE

Procedure CKWRITE copies a logical record from the program's data area to an output or an input/output KSAM file.

```
CALL "CKWRITE" USING filetable, status, record, recordsize
```

A call to procedure CKWRITE may be used to write records to a KSAM file either in sequential order or randomly by key value. The file must have been opened for output or for input/output, but not for input only.

Parameters

filetable — An 8 halfword record containing the number and name of the file, its input/output type, access mode, and a code indicating whether the previous operation on the file was successful and if so what, it was.

status — One halfword (two 8-bit characters) set to a pair of values upon completion of the call to CKWRITE to indicate whether or not the record was successfully written and if not, why not.

record — A record defined in the WORKING-STORAGE SECTION containing data to be written to the file by CKWRITE.

recordsize — An integer (S9(4)COMP) containing the length in characters of the record to be written. It must not exceed the maximum record length established for the file when it was created, and it must be long enough to contain all the keys.

Operation Notes

The file to which the content of *record* is written must be open for output only if sequential mode is specified. It may be opened for output or input/output if the access mode at open is random or dynamic.

When the file is opened for sequential access (access mode = 0) and for output only (I-O type = 1), then records must be written to the file in ascending sequential order by primary key value. The value of the primary key in the record to be written must be greater than the value of the primary key in any record previously written to the file. This ensures that the records written to the file are initially in ascending order physically as well as logically.

When I-O type = 1, CKWRITE writes records starting at the beginning of the file, thereby effectively clearing any records previously written to the file.

In a file opened for random or dynamic access (access mode = 1 or 2) and for output only or for input/output (I-O type = 1 or 2), records can be written in any order. The value of the primary key need not be in any particular relation to the primary key values of previously written records.

If you want to preserve existing records in the file, you should open the file with the input/output type equal to 2; when input/output type = 1, all existing records are cleared prior to the write.

If the file was opened for shared access with `CKOPENSHR`, then you must lock the file with a call to `CKLOCK` before writing any records. After the records are written, you should unlock the file with a call to `CKUNLOCK`.

The invalid key condition (left byte of *status*=2) can occur as a result of the following circumstances:

- File was opened for sequential access in output mode and the value of the primary key in the record being written is less than or equal to the value of the primary key in the record just written; *status*=21.
- File was opened for sequential or random access in output or input/output mode and the value of the primary key is equal to the value of the primary key in an existing record; *status*=22.
- File was opened for sequential or random access in output or input/output mode and the value of an alternate key for which duplicates are prohibited equals the value of a corresponding key in an existing record; *status*=22.
- File was opened for sequential or random access in output or input/output mode and an attempt was made to write a record beyond the physical bounds of the file; *status*=24.

Assume a KSAM file called `KSAMFILE` with records containing 74 characters, one primary key containing a name, and an alternate key containing a phone number. The data is read from an input file called `DATA-FILE`. (Refer to Figure A-2 for a diagram of the structure of this file.)

The first example writes data to `KSAMFILE` in sequential order by the primary key.

```
DATA DIVISION
⋮
WORKING-STORAGE SECTION.
77  RECSIZE        PIC S9(4)     COMP VALUE 74.
77  RESULT         PIC 9(4)      VALUE 0.
01  REC.
  03  FILLER       PIC XX        VALUE SPACES.
  03  NAME         PIC X(20).
  03  PHONE        PIC X(8).
  03  OTHERDATA    PIC X(44).
01  DAT.
  03  NAME         PIC X(20).
  03  PHONE        PIC X(8).
  03  OTHERDATA    PIC X(44).
01  FILETABLE.
  03  FILENUMBER   PIC S9(4)      COMP VALUE 0.
  03  FILENAME     PIC X(8)       VALUE "KSAMFILE".
  03  I-O-TYPE     PIC S9(4)      COMP VALUE 0.
  03  A-MODE       PIC S9(4)      COMP VALUE 0.
  03  PREV-OP      PIC S9(4)      COMP VALUE 0.
01 STAT.
  03  STATUS-KEY-1 PIC X.
  03  STATUS-KEY-2 PIC X.
⋮
PROCEDURE DIVISION.
START.
⋮
```

```
    MOVE 1 TO I-O-TYPE,<--- set type to output only
    CALL "CKOPEN" USING FILETABLE, STAT.
    IF STATUS-KEY-1="0" THEN GO TO WRITE-F.
    DISPLAY "CKOPEN ERROR, STATUS = ", STAT.
    IF STATUS-KEY-1= "9" THEN
       CALL "CKERROR" USING STAT, RESULT
       DISPLAY "CKERROR NO. ", RESULT.
    STOP RUN.
WRITE-F.
    READ DATA-FILE INTO DAT;
       AT END GO TO FINISH.
    MOVE CORRESPONDING DAT TO REC.
    CALL "CKWRITE" USING FILETABLE, STAT, REC, RECSIZE.
    IF STATUS-KEY-1="0" THEN
       DISPLAY REC.
       GO TO WRITE-F.
    IF STAT="21" THEN
       DISPLAY "SEQUENCE ERROR IN", NAME OF REC
       GO TO WRITE-F.
    IF STAT = "22" THEN
       DISPLAY "DUPLICATE KEY", NAME OF REC
       GO TO WRITE-F.
    IF STAT = "24" THEN
       DISPLAY "END OF FILE"
       GO TO FINISH.
⋮
FINISH
    CLOSE DATA-FILE.
    CALL "CKCLOSE" USING FILETABLE, STAT.
    IF STATUS-KEY-1="9" THEN
       CALL "CKERROR" USING STAT, RESULT
       DISPLAY "CKCLOSE ERROR NO. ", RESULT.
    STOP RUN.
```

The second example, using the same DATA DIVISION and the same FINISH procedure, writes one record to the file containing "ADAMSON JOHN" as its primary key value.

```
PROCEDURE DIVISION.
START.
⋮
  MOVE 1 TO I-O TYPE.<--- output only
  MOVE 2 TO A-MODE.<--- random access
  CALL "CKOPEN"USING FILETABLE, STAT.
⋮ check status
FIND-REC.
  READ DATA-FILE INTO DAT;
     AT END GO TO FINISH.
  IF NAME OF DAT = "ADAMSON        JOHN" THEN
     GO TO WRlTE-REC;
     ELSE GO TO FIND-REC.
WRITE-REC.
  MOVE CORRESPONDING DAT TO REC.
  CALL "CKWRITE" USING FILETABLE, STAT, REC, RECSIZE.
  IF STATUS-KEY-1="O" THEN
     DISPLAY REC," RECORD WRITTEN"
     GO TO FINISH.
  IF STAT = "22" THEN
     DISPLAY "DUPLICATE KEY"
     GO TO FINISH.
  IF STAT = "24" THEN
     DISPLAY "NO ROOM IN FILE"
     GO TO FINISH.
```

Examples of KSAM File Access

The following three examples illustrate KSAM file access from a COBOL program. The file accessed in each example is called KSAMFILE. It was created previously with BYTE type keys: the primary key containing the name of a person and the alternate key containing his telephone number. The remaining data in each record is his address.

Sequential Write

The first example reads data from an input file into working storage and then writes it to a KSAM file. Access mode is sequential so that as each record is written, the keys are linked in sequential order although the records are not physically written in sequence. Input/output type is output only, the only type allowed for the procedure `CKWRITE`. The following procedures are illustrated:

- `CKOPEN`
- `CKWRITE`
- `CKCLOSE`

```
Input to EXAMP1:

NOLAN     JACK   923-4975  967 REED AVE.         SUNNYVALE   CA. 94087
HOSODA    JOE    227-8214  1180 SAINT PETER CT.  LOS ALTOS   CA. 94022
ECKSTEIN  LEO    287-5137  5303 STEVENS CREEK    SANTA CLARA CA. 95050
CARDIN    RICK   578-7018  11100 WOLFE ROAD      CUPERTINO   CA. 94053
PASBY     LINDA  295-1187  TOWN & CNTRY VILLAGE  SAN JOSE    CA. 94012
SEELY     HENRY  293-4220  1144 LIBERTY ST.      EL CERRITO  CA. 94053
ROBERT    GERRY  258-5535  12345 TELEGRAPH AVE.  BERKELEY    CA. 90871
TURNEWR   IVAN   984-8498  22905 EMERSON ST.     OAKLAND     CA. 98234
WHITE     GORDON 398-0301  4350 ASHBY AVE.       BERKELEY    CA. 91234
WESTER    ELDER  287-4598  1256 KINGFISHER ST.   SUNNYVALE   CA. 43098
**END OF INPUT FOR EXAMP1**

Program EXAMP1
001000  IDENTIFICATION DIVISION.
001100  PROGRAM-ID.  EXAMP1.
001200  ENVIRONMENT  DIVISION.
001300  INPUT-OUTPUT SECTIONS
001400  FILE-CONTROL.
001500     SELECT SEQ-DATA ASSIGN TO "SEQDATA".
001600  DATA DIVISION.
001700  FILE SECTION.
001800  FD SEQ-DATA
001900     LABEL RECORDS ARE STANDARD.
002000  01 INPUT-REC.
002100     05  REAL-DATA      PIC  X(72).
002200  WORKING-STORAGE SECTION.
002300  77  RECSIZE           PIC  S9(4) COMP  VALUE 74.
002400  77  RESULT            PIC 9(4)         VALUE ZERO.
002500  01  DATA-REC.
002600      05  FILLER        PIC XX VALUE SPACES.
002700      05  REAL-DATA     PIC X(72).
002800  01  FILETABLE.
002900      02  FILENUMBER    PIC S9(4)   COMP VALUE 0.
003000      02  FILENAME      PIC X(8)         VALUE "KSAMFILE".
003100      02  I-O-TYPE      PIC S9(4)   COMP VALUE 1.
003200      02  A-MODE        PIC S9(4)   COMP VALUE 0.
003300      02  PREV-OP       PIC S9(4)   COMP VALUE 0.
003400  01  STATUSKEY.
003500      02  STATUS-KEY-1  PIC X.
003600      02  STATUS.KEY-2  PIC X.
003700
```

Figure A-4. Sequential Write Using COBOL

```
003800  PROCEDURE DIVISION.
003900  START.
004000     OPEN INPUT SEQ-DATA
004100     CALL "CKOPEN" USING FILETABLE, STATUSKEY.
004200     IF STATUS-KEY-1="9" THEN
004300        CALL "CKERROR" USING STATUSKEY, RESULT
004400        DISPLAY "CKOPEN ERROR NO.", RESULT.
004500     IF STATUS-KEY-1 NOT = "0" THEN
004600        DISPLAY "CKOPEN FAILED"
004700        STOP RUN.
004800  LOOP.
004900     READ SEQ-DATA
005000        AT END GO TO FINISH.
005100     MOVE CORP INPUT-REC TO DATA-REC.
005200     CALL "CKWRITE" USING FILETABLE, STATUSKEY, DATA-REC,
005300        RECSIZE.
005400     IF STATUSKEY = "02" THEN
005500        DISPLAY "DUPLICATE KEY".
005600     IF STATUS-KEY-1 = "0" THEN
005700        DISPLAY DATA-REC
005800        GO TO LOOP.
005900     IF STATUS-KEY-1 = "9" THEN
006000        CALL "CKERROR" USING STATUSKEY, RESULT
006100        DISPLAY "CKWRITE ERROR NO.", RESULT
006200        DISPLAY DATA-REC
006300        GO TO LOOP.
006400  FINISH.
006500     CLOSE SEQ-DATA.
006600     CALL "CKCLOSE" USING FILETABLE, STATUSKEY.
006700     IF STATUS-KEY-1 = "9" THEN
006800        CALL "CKERROR" USING STATUSKEY, RESULT
006900        DISPLAY "CKCLOSE ERROR NO. ". RESULT.
007000     STOP RUN.

Output from EXAMP1 Execution:

NOLAN     JACK    923-4975   967 REED AVE.          SUNNYVALE    CA.  94087
HOSODA    JOE     227-8214   1180 SAINT PETER CT.   LOS ALTOS    CA.  94022
ECKSTEIN  LEO     287-5137   5303 STEVENS CREEK     SANTA CLARA  CA.  95050
CARDIN    RICK    578-7018   11100 WOLFE ROAD       CUPERTINO    CA.  94053
PASBY     LINDA   295-1187   TOWN & CNTRY VILLAGE   SAN JOSE     CA.  94012
SEELY     HENRY   293-4220   1144 LIBERTY ST.       EL CERRITO   CA.  94053
ROBERT    GERRY   258-5535   12345 TELEGRAPH AVE .  BERKELEY     CA.  90871
TURNEWR   IVAN    984-8498   22905 EMERSON ST.      OAKLAND      CA.  98234
WHITE     GORDON  398-0301   4350 ASHBY AVE.        BERKELEY     CA.  91234
WESTER    ELDER   287-4598   1256 KINGFISHER ST.    SUNNYVALE    CA.  43098
END OF PROGRAM
```

Sequential Write Using COBOL(continued)

Sequential Read

The second example reads the file KSAMFILE in sequential order by primary key (NAME) and prints each record as it is read. It then repositions the file to the first sequential record according to the alternate key (PHONE) and prints each of the records as it is read in this order. The file is opened in sequential mode for input only. The following procedures are illustrated:

- CKOPEN
- CKREAD
- CKSTART
- CKCLOSE

```
Program EXAMP2:

001000  IDENTIFICATION DIVISION.
001100  PROGRAM-ID. EXAMP2.
001200  ENVIRONMENT DIVISION.
001300  INPUT-OUTPUT SECTION.
001400  FILE-CONTROL.
001500      SELECT SEQ-DATA ASSIGN TO "SEQDATA".
001600  DATA DIVISION.
001700  WORKING-STORAGE SECTION.
001800  77  RECSIZE          PIC  S9(4)  COMP  VALUE 74.
001900  77  RESULT           PIC 9(4)          VALUE  ZERO.
002000  77  KEY-LOC          PIC S9(4)   COMP  VALUE 23.
002100  77  RELOP            PIC S9(4)   COMP  VALUE 2.
002200  77  KEYLENGTH        PIC S9(4)   COMP  VALUE 8.
002300  77  KEY-VALUE        PIC X(8)          VALUE "000-0000".
002400  01  DATA-REC.
002500      05  FILLER       PIC XX.
002600      05  NAME         PIC X(20).
002700      05  PHONE        PIC X(8).
002800      05  OTHER-DATA   PIC X(44).
002900 01   FILETABLE.
003000      02  FILENUMBER   PIC S9(4)   COMP VALUE o.
003100      02  FILENAME     PIC X(8)         VALUE "KSAMFILE".
003200      02  I-O-TYPE     PIC S9(4)   COMP VALUE o.
003300      02  A-MODE       PIC S9(4)   COMP VALUE o.
003400      02  PREV-OP      PIC S9(4)   COMP VALUE o.
003500  01  STATUSKEY.
003600      02  STATUS-KEY-1 PIC X.
003700      02  STATUS-KEY-2 PIC X.
003800
```

Figure A-5. Sequential Read Using COBOL

```
003900  PROCEDURE DIVISION.
004000  START.
004100     CALL "CKOPEN" USING FILETABLE, STATUSKEY.
004200     IF STATUS-KEY-1 = "9" THEN
004300        CALL "CKERROR" USING STATUSKEY, RESULT
004400        DISPLAY "CKOPEN ERROR NO.", RESULT.
004500     IF STATUS-KEY-1 NOT = "0" THEN
004600        DISPLAY "CKOPEN FAILED"
004700        STOP RUN.
004800     DISPLAY "ALPHABETICAL ORDER"
004900     DISPLAY " ".
005000  LOOP1.
005100     CALL "CKREAD" USING FILETABLE, STATUSKEY, DATA-REC,
005200        RESIZED.
005300     IF STATUS-KEY-1= "1" THEN GO TO PART2.
005400     IF STATUS-KEY-1 = "0" THEN
005500        DISPLAY DATA-REC
005600     ELSE
005700        DISPLAY "CKREAD ERROR, STATUS = ", STATUSKEY
005800        IF STATUS-KEY-1 = "9" THEN
005900           CALL "CKERROR" USING STATUSKEY, RESULT
006000           DISPLAY "ERROR NO.", RESULT.
006100     GO TO LOOP.
006200  PART2.
006300     DISPLAY " ".
006400     DISPLAY "PHONE NO. ORDER:"
006500     DISPLAY " ".
006600     CALL "CKSTART" USING FILETABLE, STATUSKEY, RELOP,
006700        KEY-VALUE, KEY-LOC, KEYLENGTH.
006800     IF STATUSKEY = "23" THEN GO TO FINISH.
006900     IF STATUS-KEY-1 = "0" THEN GO TO LOOP2.
007000     DISPLAY "CKSTART ERROR, STATUS = ", STATUSKEY.
007100     IF STATUS-KEY-1 = "9" THEN
007200        CALL "CKERROR" USING STATUSKEY, RESULT
007300        DISPLAY "ERROR NO.", RESULT.
007400     GO TO FINISH.
```

Sequential Read Using COBOL (continued)

```
007500  LOOP2.
007600     CALL "CKREAD" USING FILETABLE, STATUSKEY, DATA-REC,
007700        RECSIZE.
007800     IF STATUS-KEY-1 = "1" THEN GO TO FINISH.
007900     IF STATUS-KEY-1 = "0" THEN
008000        DISPLAY DATA-REC
008100     ELSE
008200        DISPLAY "CKREAD ERROR, STATUS =", STATUSKEY
008400     IF STATUS-KEY-1 ="9" THEN
008400        CALL "CKERROR" USING STATUSKEY, RESULT
008500        DISPLAY "ERROR NO. ", RESULT.
008600     GO TO LOOP2.
008700  FINISH.
008800     CALL "CKCLOSE" USING FILETABLE, STATUSKEY.
008900     IF STATUS-KEY-1 = "9" THEN
009000        CALL "CKERROR" USING STATUSKEY, RESULT
009100        DISPLAY "CKCLOSE ERROR NO.", RESULT.
009200      STOP RUN.

Output from EXAMP2 Execution:
ALPHABETICAL ORDER:
CARDIN    RICK    587-7018  11100 WOLFE ROAD      CUPERTINO CA.    94053
ECKSTEIN  LEO     287-5137  5303 STEVENS CREEK    SANTA CLARA CA.  95050
HOSODA    JOE     227-8214  1180 SAINT PETER CT.  LOS ALTOS CA.    94022
NOLAN     JACK    923-4975  967 REED AVE.         SUNNYVALE CA.    94087
PASBY     LINDA   295-1187  TOWN & CNTRY VILLAGE  SAN JOSE CA.     94102
ROBERT    GERRY   259-5535  12345 TELEGRAPH AVE.  BERKELEY CA.     90871
SEELY     HENRY   293-4220  1144 LIBERTY ST.      EL CERRITO CA.   94053
TURNEWR   IVAN    984-8498  22905 EMERSON ST.     OAKLAND CA.      98234
WESTER    ELDER   287-4598  1256 KINGFISHER ST.   SUNNYVALE CA.    43098
WHITE     GORDON  398-0301  4350 ASHBY AVE.       BERKELEY CA.     91234
```

Sequential Read Using COBOL (continued)

```
PHONE NO. ORDER:
HOSODA    JOE     227-8214  1180 SAINT PETER CT.  LOS ALTOS CA.   94022
ROBERT    GERRY   259-5535  12345 TELEGRAPH AVE.  BERKELEY CA.    90871
WESTER    ELDER   287-4598  1256 KINGFISHER ST.   SUNNYVALE CA.   43098
ECKSTEIN  LEO     287-5137  5303 STEVENS CREEK    SANTA CLARA CA. 95050
SEELY     HENRY   293-4220  1144 LIBERTY ST.      EL CERRITO CA.  94053
PASBY     LINDA   295-1187  TOWN & CNTRY VILLAGE  SAN JOSE CA.    94102
WHITE     GORDON  398-0301  4350 ASHBY AVE.       BERKELEY CA.    91234
CARDIN    RICK    578-7018  11100 WOLFE ROAD      CUPERTINO CA.   94053
NOLAN     JACK    923-4975  967 REED AVE.         SUNNYVALE CA.   94087
TURNEWR   IVAN    984-8498  22905 EMERSON ST.     OAKLAND CA.     98234

END OF PROGRAM
```

Sequential Read Using COBOL (continued)

Random Update

This example reads a set of new data containing update information into the WORKING-STORAGE SECTION. Each record read is followed by a U for update, a D for delete, or an A for add. Records to be added are written to the file `KSAMFILE` using `CKWRITE` in random mode. Records to be updated are copied to the appropriate record with `CKREWRITE`. Records to be deleted are first read into the WORKING-STORAGE SECTION with `CKREADBYKEY` and then deleted with `CKDELETE`. The file is opened in random mode for input/output.

The procedures illustrated by this example are:

- `CKOPEN`
- `CKREADBYKEY`
- `CKDELETE`
- `CKREWRITE`
- `CKWRITE`
- `CKCLOSE`

```
Program EXAMP3:

001000  IDENTIFICATION DIVISION,
001100  PROGRAM-ID. EXAMP3.
001200  ENVIRONMENT DIVISION.
001300  INPUT-OUTPUT SECTION.
001400  FILE-CONTROL.
001500      SELECT NEW-DATA ASSIGN TO "NEWDATA".
001600  DATA DIVISION.
001700  FILE SECTION.
001800  FD  NEW-DATA
001900      LABEL RECORDS ARE STANDARD.
002000  01  INPUT-REC            PIC X(73),
002100  WORKING-STORAGE SECTION,
002200  77  RECSIZE              PIC S9(4)  COMP VALUE 74.
002300  77  RESULT               PIC 9(4)        VALUE ZERO.
002400  77  KEY-LOC              PIC S9(4)  COMP VALUE 3.
002500  01  MASTER-REC.
002600      05  FILLER           PIC XX.
002700      05  NAME             PIC X(20).
002800      05  PHONE            PIC X(8).
002900      05  OTHER-DATA       PIC X(44).
003000  01  DATA-REC.
003100      05  NAME             PIC X(20).
003200      05  PHONE            PIC X (8).
003300      05  OTHER-DATA       PIC X(44).
003400      05  TRANSACTION-CODE PIC X.
003500  01  FILETABLE.
003600      02  FILENUMRER       PIC S9(4)  COMP VALUE o.
003700      02  FILENAME         PIC X(8)        VALUE "KSAMFILE".
003800      02  I-O-TYPE         PIC S9(4)  COMP VALUE 2.
003900      02  A-MoDE           PIC S9(4)  COMP VALUE 1.
004000      02  PHEV-OP          PIC S9(4)  COMP VALUE 0.
004100  01  STATUSKEY.
004200      02  STATUS-KEY-1     PIC X.
004300      02  STATUS-KEY-2     PIC X.
004400
```

Figure A-6. Random Update with COBOL

```
004500  PROCEDURE DIVISION.
004600  START.
004700    OPEN INPUT NEW-DATA.
004800    CALL "CKOPEN" USING FILETABLE, STATUSKEY.
004900    IF STATUS-KEY-1 = "9" THEN
005000       CALL "CKERROR" USING STATUSKEY, RESULT
005100       DISPLAY "CKOPEN ERROR NO.", RESULT.
005200    IF STATUS-KEY-1 NOT ="0" THEN
005300       DISPLAY "CKOPEN FAILED"
005400       STOP RUN.
005500  LOOP.
005600    READ NEW-DATA INTO DATA-REC;
005700       AT END GO TO FINISH.
005800    IF TRANSACTION-CODE = "A" THEN GO TO ADD-REC,
005900    IF TRANSACTION-CODE NOT = "D" AND "U" THEN
006000       DISPLAY "ILLEGAL TRANSACTION CODE"
006100       DISPLAY DATA-REC
006200       GO TO LOOP.
006300    CALL "CKREADBYKEY" USING FILETABLE, STATUSKEY, MASTER-REC,
006400       NAME OF DATA-REC, KEY-LOC, RECSIZE.
006500    IF STATUS-KEY-1 NOT = "0" THEN
006600       DISPLAY "CKREADBYKEY ERROR, STATUS =", STATUSKEY,
006700          ";  KEY =", NAME OF DATA-REC
006800       IF STATUS-KEY-1 = "9" THEN
006900          CALL "CKERROR" USING STATUSKEY, RESULT
007000          DISPLAY "ERROR NO.", RESULT
007100          GO TO LOOP
007200       ELSE
007300          GO TO LOOP.
007400    IF TRANSACTION-CODE = "D" THEN GO TO DELETE-REC.
007500    MOVE CORR DATA-REC TO MASTER-REC.
007600    CALL "CKREWRITE" USING FILETABLE, STATUSKEY, MASTER-REC,
007700       RECSIZE.
007800    IF STATUS-KEY-1 = "0" THEN
007900       DISPLAY MASTER-REC, "UPDATED"
008000       GO TO LOOP.
008100    DISPLAY "CKREWRITE ERROR, STATUS =", STATUSKEY, "; KEY ="
008200       NAME OF MASTER-REC.
008300    IF STATUS KEY-1= "9" THEN
008400       CALL "CKERROR" USING STATUSKEY, RESULT
008500       DISPLAY "ERROR NO.", RESULT
008600       GO TO LOOP.
```

Random Update with COBOL (continued)

```
008700  DELETE-REC.
008800    CALL "CKDELETE" USING FILETABLE, STATUSKEY.
008900    IF STATUS-KEY-1 = "0" THEN
009000       DISPLAY MASTER-REC, "DELETED"
009100       GO TO LOOP.
009200    DISPLAY "CKDELETE ERROR, STATUS =" STATUSKEY.
009300    IF STATUS-KEY-1 = "9" THEN
009400        CALL "CKERROR", USING STATUSKEY, RESULT
009500        DISPLAY "ERROR NO.", RESULT.
009600    GO TO LOOP.
009700  ADD-REC.
009800    MOVE CORR DATA-REC TO MASTER-REC.
009900    CALL "CKWRITE" USING FILETABLE, STATUSKEY, MASTER-REC.
010000        RECSIZE.
010100    IF STATUSKEY = "02" THEN
010200        DISPLAY "DUPLICATE KEY",
010300    IF STATUS-KEY-1 = "0" THEN
010400        DISPLAY MASTER-REC, "ADDED"
010500        GO TO LOOP.
010600    DISPLAY "CKWRITE ERROR, STATUS = ", STATUSKEY.
010700    IF STATUS-KEY-1 = "9" THEN
010800        CALL "CKERROR" USING STATUSKEY, RESULT
010900        DISPLAY "ERROR NO. ", RESULT.
011000    DISPLAY MASTER-REC,
011100    GO TO LOOP.
011200  FINISH.
011300    CLOSE NEW-DATA.
011400    CALL "CKCLOSE" USING FILETABLE, STATUSKEY,
011500    IF STATUS-KEY-1 = "9" THEN
011600       CALL "CKERROR" USING STATUSKEY, RESULT
011700       DISPLAY "CKCLOSE ERROR NO.", RESULT
011800    STOP RUN.
```

Random Update with COBOL (continued)

```
Input to EXAMP3:

NOLAN      JACK     923-4975  1 ANY STREET.          SUNNYVALE CA.   94087U
SMITH      JOHN     555-1212  102 FIRST ST.          OUR TOWN CA.    94099A
ECKSTEIN   LEO                                                         D
CARDIN     RICK     257-7000  11100 WOLFE ROAD       CUPERTINO CA.   94041U
PASBY      LINDAL                                                      D
JANE       MARY     565-9090  1776 BICENTENNIAL ST.  AMAHEIM CA.     91076A
ROBERT     GERRY    259-5535  12345 TELEGRAPH AVE.   BERKELEY CA.    94704U
TURNEW     IVAN                                                        D
FORD       GERALD   555-1976  1600 PENNSYLVANIA      WASHINGTON DC.  20001U
WESTER     ELDER    287-4598  1256 KINGFISHER ST.    SUNNYVALE CA.   94309A

Output from Execution of EXAMP3:

NOLAN      JACK      923-4975  1 ANY STREET.         SUNNYVALE   CA. 94087
                                                                     UPDATED
SMITH      JOHN      555-1212  102 FIRST ST.         OUR TOWN    CA. 94099
                                                                     ADDED
ECKSTEIN   LEO       287-5137  5303 STEVENS CREEK    SANTA CLARA CA. 95050
                                                                     DELETED
CARDIN     RICK      257-7000  11100 WOLFE ROAD      CUPERTINO   CA. 94014
                                                                     UPDATED
PASBY      LINDA     295-1187  TOWN & CNTRY VILLAGE  SAN JOSE    CA. 94102
                                                                     DELETED
JANE       MARY      565-9090  1776 BICENTENNIAL ST. ANAHEIM     CA. 91076
                                                                     ADDED
ROBERT     GERRY     259-5535  12345 TELEGRAPH AVE.  BERKELEY    CA. 94704
                                                                     UPDATED
CKREADBYKEY  ERROR, STATUS = 23;  KEY = TURNEW    IVAN
CKREADBYKEY  ERROR, STATUS = 23;  KEY = FORD      GERALD
CKWRITE  ERROR, STATUS = 22
WESTER     ELDER     287-4598  1256 KINGFISHER ST.   SUNNYVALE   CA. 94309
```


Note

Note that the input contains data that results in error messages. The name IVAN TURNEW is spelled incorrectly and cannot be found. The name GERALD FORD does not exist in the original file and also cannot be found. On the other hand, the name ELDER WESTER already exists in the file and cannot be added since it is a primary key for which duplicates are not allowed.

B

BASIC/V Intrinsics

The BASIC/V interpreter and compiler require special intrinsics to access existing KSAM files. The following intrinsics were developed for these BASIC/V programs.

Note

These intrinsics are provided to allow BASIC/V programs to run in compatibility mode. Do not use these intrinsics when writing new programs in other languages or when porting BASIC/V programs. If you are porting to Business BASIC/XL, use the standard file intrinsics discussed in this manual.

Overview

KSAM files are accessed from BASIC/V programs through calls to a set of input/output procedures. These procedures allow you to open, write records to, read records from, update and delete records, position, lock, unlock, and close KSAM files.

A KSAM file must already exist before it can be accessed from a BASIC/V program. The BASIC/V procedures for accessing KSAM files do not provide a means to create a KSAM file.

The BASIC/V procedures to access KSAM files perform input/output activities differently from the BASIC/V input/output commands. The KSAM procedures read and write records in their entirety. Once part of a record has been read or written by one of the KSAM file access procedures, the entire record has, in actuality, been read or written. A subsequent call will access another record.

Character substrings are expressions when used in the BASIC/V KSAM procedures. As such, no values can be returned to them. A copy of the substring is passed as the actual parameter.

Calling a KSAM Procedure

The KSAM interface procedures are called from a BASIC program with a CALL statement of the following general form:

statementlabel `CALL` *procname* (*filenumber, status* `[,`*parameterlist*`]`)

Where:

statementlabel	The number of the statement in the program.
procname	The KSAM access procedure to which control is transferred.
filenumber	A numeric variable whose value identifies an open KSAM file. This parameter must be present. Its value is assigned when the file is opened and must not be changed until the file is closed.
status	A 4-character string variable to which a code is returned that indicates whether the current operation was successful or not, and if not, the reason for failure.
parameterlist	A set of one or more parameters that, if present, further define input/output operations on this file.

The first two parameters, *filenumber* and *status* are included in every KSAM procedure call, except `BKERROR` and `BKVERSION`. The parameters in *parameterlist* depend on the procedure in which they are used. Some *parameterlist* parameters are optional and, if omitted, default values are assigned by KSAM. Such parameters are indicated by brackets in the procedure call format. The required parameters *filenumber* and *status* are both variables, the first numeric, the second string. Other parameters are either variables or expressions. Expressions are either variables or constants, or a combination of both. The data type of the parameter depends on its definition in the procedure. The procedure call formats specify the data type of each parameter.

Depending on the procedure, certain variables can be assigned values as a result of executing the procedure. The procedure itself is never assigned a value.

Optional Parameters

When parameters in *parameterlist* are optional, those parameters are surrounded by brackets. In a series of optional parameters, the enclosing brackets are nested. For example:

`CALL name (`*filenum,status*`[,`*param1*`[,`*param2*`[,`*param3*`]]])`

This notation tells you that parameters can be omitted only from the end of the optional list; parameters cannot be omitted from the middle or beginning of the list. For example, if you want to specify *param3*, you must also specify the preceding parameters, *param1* and *param2*. If you specify *param2*, you can omit the following parameter *param3*, but not the preceding *param1*.

Status Parameter

The status parameter is a four-character string variable to which the status of the input/output operation is returned. It is the second parameter in every KSAM procedure call except BKERROR, in which it is the first parameter.

The first character of the *status* string determines its general type. The other three characters supply specific codes to further define the status. The operation of a called procedure is successful only if the first character returned in *status* is zero. Other values returned to *status* indicate the reason an operation was not successful. You can convert any status value to a printable message by calling BKERROR. By combining the two parts of the status code, the following values may be returned to the status parameter:

00 *Successful completion—*

The current input/output operation was completed successfully; no duplicate keys read or written.

02 *Successful completion; Duplicate key—*

- In a call to BKREAD or BKREADBYKEY, the current key has the same value as the equivalent key in the next sequential record; duplicate keys are allowed for the key.
- In a call to BKWRITE or BKREWRITE, the record just written created a duplicate key value for at least one key for which duplicates are allowed.

10 *At end condition—*

A sequential read was attempted with BKREAD and there was no next logical record in ascending sequence according to the primary key value or the current alternate key value. Or an attempt was made by BKSTART or BKREADBYKEY to position the pointer to a record whose key value was less than the lowest key value or higher than the highest key value.

21 *Invalid key; Sequence error—*

- In a call to BKWRITE for a file opened with sequence checking, the record being written contains a primary key that is less than a key in a previously written record.
- In a call to BKREWRITE, the primary key value was changed in the program since a successful execution of BKREAD defined the record to be rewritten.

22 *Invalid key; Duplicate key error—*

An attempt was made to write or rewrite a record with BKWRITE or BKREWRITE and the record would create a duplicate key value in a key for which duplicates are not allowed.

23 *Invalid key; No record found—*

An attempt was made to locate a record by a key value with `BKSTART` or `BKREADBYKEY` and the record cannot be found.

24 *Invalid key; Boundary violation—*

An attempt was made with `BKWRITE` to write beyond the externally defined boundaries of the file; that is, to write past the end-of-file.

71 *Request denied; File already locked—*

An attempt was made to lock a file with `BKLOCK` and the file is already locked.

81 *Invalid call; Invalid number of parameters—*

Too many or too few parameters were specified in the procedure call just made.

82 *Invalid call; Invalid parameter—*

The specified parameter is not the correct type. For example, a string variable was selected where only a numeric variable or expression is allowed.

83 *Invalid call; Insufficient internal buffer space—*

The data specified in the *parameterlist* to be read or written will not fit into the configured internal buffer space. You may need to have certain operating system parameters revalued.

9xxx *File system error—*

An MPE file system error occurred for which the three-character value, *xxx* is the error code. You can call procedure `BKERROR` to convert the error code returned here to a printable message.

The value of status can be tested as a whole, or the first character can be tested separately from the remaining characters. For example:

```
  10 DIM S$(4)
  :
  50 IF S$(1;1) = "0" THEN PRINT "SUCCESS"
  60 ELSE PRINT "ERRORCODE=";S$
  :
100 IF S$(1;1)= "9" THEN DO
110 PRINT "FILE ERROR=";S$(2)
120 DOEND
:
200 IFS$ = "22" THEN DO
210 PRINT "DUPLICATE KEY ERROR"
220 DOEND
300 IF S$(2)= "2" THEN PRINT "DUPLICATE KEY"
```

For any status value, you can call the BKERROR procedure and a message is returned that gives the meaning of the status code. You can then print this message rather than writing your own.

KSAM Logical Record Pointer

Many of the KSAM procedures use a *logical record pointer* to indicate the current record in the file. This pointer points to a key value in the index area that identifies the current record in the data area. The particular key used, if the file has more than one key, is the key last specified in the current or a previous procedure call. By default, it is the primary key.

Procedures that use pointers are either *pointer-dependent* or *pointer-independent*. Pointer-dependent procedures expect the pointer to be positioned at a particular record in order to execute properly. Pointer-independent procedures, on the other hand, execute regardless of where the pointer is positioned and, in most cases, they position the pointer.

Table B-1. Positioning the Logical Record Pointer

Procedure Name	Pointer-Dependent	Position of Pointer After Execution of Procedure
BKSTART	NO	Points to key whose value was specified in call.
BKREADBYKEY	NO	Points to key whose value was specified in call.
BKWRITE	NO	Points to key whose value is next in ascending key sequence to key value in record just written.
BKREAD	YES	Pointer remains positioned to key value for record just read; unless the next call is to BKREAD, or to BKREWRITE followed by BKREAD, in which case, the pointer is moved to the next record in key sequence before the read.
BKDELETE	YES	Points to next key value in ascending sequence following key value in record just deleted.
BKREWRITE	YES	Pointer remains positioned to key value for record just modified; *unless* any key value in record was changed, in which case, it points to next key in ascending sequence after the key in the modified record.

BASIC procedures do not access a KSAM file in physical sequence or by record number; they ignore the physical pointer.

Shared Access

Particular care must be taken when using the logical record pointer during shared access. Since the record pointer is maintained in a separate control block for each open file, one user may cause the record pointer to be inaccurate without other users being aware of it. To avoid this problem, you should always lock the file in a shared environment before calling any procedure that sets the pointer and leave the file locked until all procedures that depend on that pointer have been executed. Thus, if you want to read the file sequentially, delete a record, or modify a record, you should lock the file, call a procedure that sets the pointer (such as BKSTART), and then call BKREAD, BKDELETE, or BKREWRITE. When the operation is complete, you can then unlock the file to give other users access to it.

BKCLOSE

A call to BKCLOSE terminates file processing for the specified KSAM file.

```
CALL BKCLOSE (filenum, status)
```

When processing is completed, a KSAM file should be closed with a call to BKCLOSE. No further processing is allowed on the file until a BKOPEN procedure call reopens the file.

BKCLOSE can be executed only for a file that is open.

Parameters

filenum — A numeric variable containing the file number that identifies the file; this number was returned by the last call to BKOPEN. It should not be altered until the file is closed with a successful call to BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the file was successfully closed and if not, why not. The first character is set to 0 if the close is successful, to another value if not. (*Required parameter*)

Operation Notes

After calling BKCLOSE, you should check the *status* parameter to determine if the file was closed successfully. A successfully closed file is no longer available for processing until it is reopened. Note that a KSAM file can be closed and then reopened in order to specify a different access mode or type of processing.

The BKCLOSE procedure does not remove the file from the system. To do this, you should use the PURGE command of KSAMUTIL or MPE/iX.

The example in Figure B-1 closes a file identified by the file number in F. It then checks the status and prints a message if the status shows any code except the zero for successful completion.

```
3610 REM ********************************************************
3620 REM * CLOSE A KSAM FILE *
3630 REM ********************************************************
3640 REM
3650 REM F IS THE FILE NUMBER OF A KSAM FILE
3660 REM DEFINED BY A CALL TO BKOPEN
3670 REM
3680 CALL BKCLOSE(F,S$)
3690 REM
3700 REM NOW DETERMINE WHETHER THIS CALL SUCCEEDED
3710 REM
3720 IF S$[1,1]<>"0" THEN DO
3730 REM N$ CONTAINS THE NAME OF THE KSAM FILE
3740 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
3750 PRINT "UNABLE TO CLOSE ";N$;" ERROR ";S$[1;1];" DETAIL ";S$[2]
3760 CALL BKERROR(S$,M$)
3770 PRINT M$
3780 DOEND
```

Figure B-1. Closing a KSAM File with BKCLOSE

BKDELETE

Logically deletes a record from a KSAM file.

```
CALL BKDELETE (filenum, status)
```

A call to BKDELETE logically deletes the record referenced by the logical record pointer. If reuse is not specified, then a logically deleted record is marked for deletion, but is not physically removed from the file. The connection between a data record marked for deletion and the index area is severed.

When a file with deleted records is copied by FCOPY to a new KSAM file, records marked for deletion by BKDELETE are not copied. This use of FCOPY provides a means to compact a file in which many records have been marked for deletion but physically use space in the file.

To use BKDELETE, the file must be open in the access mode that allows update. If access is shared, the file must also be opened with dynamic locking allowed (*lock*=1), and the file must be locked by BKLOCK before records are deleted.

Parameters

filenum — A numeric variable containing the file number that identifies the file; this number was returned by the last call to BKOPEN. It should not be altered unless the file is closed with a successful call to BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKREWRITE was successful and if not, why not. The first character is set to zero if the call succeeds, to another value if not.

Operation Notes

Before calling BKDELETE, you can read the record to be deleted from the KSAM file into the BASIC program. Using either BKREAD or BKREADBYKEY, read the record into variables named in the read call. When BKDELETE is successfully executed, the record is marked for deletion. If reuse is not specified, then a logically deleted record is marked for deletion, but is not physically removed from the file. Any connections between the record and key entries in the index area are severed. The associated key entries are physically deleted from the index area although the data record remains in the data area. Data space is not reused in order to maintain the chronological order of the file. Because BKDELETE requires that the record be both read and written, you must open the file for update (*access* = 4) before calling this procedure.

After calling BKDELETE, you should check the *status* parameter to make sure that the delete was successful.

FCOPY can also be used to permanently remove any records that were logically deleted with BKDELETE. When you use FCOPY to copy your KSAM file to a newly created KSAM file, only active records

are copied. Records marked for deletion are dropped from the data area during the copy. The new file is more compact, particularly if many records had been deleted from the old file.

When access is shared, the call that positions the pointer to the record to be deleted should be included in the same pair of `BKLOCK/BKUNLOCK` calls as the call to `BKDELETE`. This ensures that no other user alters the record position between the call that locates the record and the call that deletes it.

Figure B-2 contains an example illustrating the logical deletion of a record from a KSAM file.

```
3240 REM ******************************************************
3250 REM * REMOVE A RECORD FROM A KSAM FILE *
3260 REM ******************************************************
3270 REM
3280 REM F IS THE FILE NUMBER OF A KSAM FILE OPENED BY A CALL TO BKOPEN
3290 REM NOTE THAT FOR BKDELETE, BKOPEN ACCESS MODE MUST = 4 FOR UPDATE
3295 REM
3300 REM THE RECORD TO BE DELETED MUST FIRST BE READ...
3305 REM AN ASSUMPTION HAS BEEN MADE THAT THE RECORD TO BE READ
3310 REM AND DELETED CONTAINS THE SAME INFORMATION THAT WAS
3320 REM WRITTEN IN THE BKWRITE EXAMPLE.
3330 REM
3340 CALL BKREAD(F,S$,B1$,B2$,A5[*],A3[*],A2[*])
3350 REM
3360 REM NOW DETERMINE WHETHER THE CALL WAS SUCCESSFUL
3370 REM
3380 IF S$[1;1]<>"0" THEN DO
3390 REM N$ CONTAINS THE NAME OF THE KSAM FILE
3400 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
3410 PRINT "UNABLE TO READ ";N$" ERROR ";S$[1;1];" DETAIL ";S$[2]
3420 CALL BKERROR(S$,M$)
3430 PRINT M$
3435 GOTO 3620
3440 DOEND
3450 REM
3460 CALL BKDELETE(F,S$)
3470 REM
3480 REM NOW DETERMINE WHETHER THIS CALL SUCCEEDED
3490 REM
3500 IF S$[1;1]<>"0" THEN DO
3510 REM N$ CONTAINS THE NAME OF THE KSAM FILE
3520 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
3530 PRINT "UNABLE TO DELETE RECORD FROM ";N$;
3535 PRINT "ERROR ";S$[1;1];"DETAIL ";S$[2]
3540 CALL BKERROR(S$,M$)
3550 PRINT M$
3560 GOTO 3620
3570 DOEND
3575 PRINT "DELETED RECORD CONTAINS ";B1$;B2$;
3576 MAT PRINT A5
3577 MAT PRINT A3,A2
3580 REM
3590 REM THE PROGRAM CONTINUES
```

Figure B-2. Deleting a Record With BKDELETE

BKERROR

A call to BKERROR returns a message corresponding to the *status* value.

CALL BKERROR (*status, message*)

Call this procedure in order to get a printable string of characters that describes the condition that corresponds to the value of the *status* parameter. The string of ASCII characters returned in *message* can be printed as an error message.

Parameters

status — A four-character string variable to which is returned a numeric value in printable form following execution of any of the procedures described in this section. The value in *status* is used to derive the text in *message*. (*Required parameter*)

message — A string variable which will contain the text describing the error whose code has been returned to *status*. This parameter should be dimensioned to at least 72 characters in length. If the message length exceeds the dimensioned length of *message*, a truncated text is provided. (*Required parameter*)

Operation Notes

The following example illustrates the use of BKERROR. Two strings are dimensioned for *message*; one (M$) is sufficiently long, the other (N$) causes truncation of the message. Assume that the status code in S$ is the value 22.

```
    10 DIM S$(4),M$(72),N$(24)
    20 REM..S$ IS THE STATUS STRING
    30 REM..M$ IS A SUFFICIENTLY LARGE STRING
    40 REM..N$ IS TOO SMALL FOR THE MESSAGE
    50 REM..ASSUME S$ CONTAINS THE VALUE "22"
    60 REM..
     :
   100 CALL BKERROR (S$,MS)
   110 PRINT "ERROR";S$(1;1);"DETAIL";S$(2);"";M$
   120 CALL BKERROR (S$,M$)
   130 PRINT "ERROR "S$(1;1);"DETAIL";S$(2);"";N$
RUN
ERROR 2 DETAIL 2 INVALID KEY VALUE. DUPLICATED KEY VALUE
ERROR 2 DETAIL 2 INVALID KEY VALUE. DUPL
```

In another example, BKERROR is called to retrieve the message corresponding to the MPE file system error code returned when the first character of status is 9.

```
10 DIM S$(4),M$(72)
:
50 IF S$(1;1)="9" THEN DO
60 CALL BKERROR(S$,M$)
70 PRINT"FILE ERROR";S$(2);"MEANS";M$
80 DOEND
```

Suppose the value returned in *status* is 9172. The routine above prints the following message when the program is run:

```
FILE ERROR 172 MEANS KEY NOT FOUND; NO SUCH KEY VALUE
```

BKLOCK

Dynamically locks KSAM file during shared access.

```
CALL BKLOCK(filenum,status[,condition])
```

When more than one user accesses the same file, BKLOCK can be used to make access to the file exclusive for one user while he writes to or updates the file. In order to use BKLOCK, the file must be opened with dynamic locking allowed by all users who are sharing the file. When finished with the changes that required exclusive access, the user who has locked the file with BKLOCK should unlock it with BKUNLOCK.

Note

Note that a file opened for shared access must be locked by BKLOCK before the file can be modified by BKWRITE, BKREWRITE, or BKDELETE.

Parameters

filenum — A numeric variable containing the file number that identifies the file; this number was returned to *filenum* by the last call to BKOPEN. It should not be altered unless the file is successfully closed by BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKLOCK was successful and if not, why not. The first character is set to zero when the call succeeds, to another value if it fails. (*Required parameter*)

condition — A numeric expression whose value determines the action taken if the file is locked by another user when BKLOCK is executed. If the value of *condition* is:

- Zero-locking is unconditional.

  If the file cannot be locked immediately because another user has locked it, your program suspends execution until the file can be locked. (*default value*)

- Non-zero-locking is conditional.

  If the file is already locked, control returns immediately to your program with *status* set to 71.

(*Optional parameter*) *Default: If omitted, locking is unconditional.*

Operation Notes

In order to call BKLOCK, the file must be opened with dynamic locking allowed. That is, the parameter *lock* in the BKOPEN procedure must be set to 1. Also, since dynamic locking is useful only when access is shared, probably the file will have been opened with the *exclusive* parameter in BKOPEN set to 3.

Users who share the same file should cooperate on how they will share the file. Unless they all agree to allow locking, no one will be able to lock the file. Also, it is important to avoid situations where one user locks the file and forgets to unlock it. If this occurs when *condition* is set to a non-zero value, the calling process is not halted. But if the file is locked already and you attempt to lock a file with *condition* omitted or set to zero, your process is halted until the other user either unlocks the file or logs off.

You should always check the *status* parameter immediately following a call to BKLOCK in order to determine if the call was completed successfully. If you locked with *condition* set to a nonzero value, you should check if the file was locked before continuing. If it was locked, status will have a 0 in the first character, but if another user had locked the file preventing your call to BKLOCK from working, then status contains the value 71.

Figure B-3 contains an example of locking a file with BKLOCK.

```
 830 REM ******************************************************
 840 REM * LOCK A KSAM FILE *
 850 REM ******************************************************
 855 REM
 860 REM F IS THE FILE NUMBER OF A KSAM FILE
 870 REM OPENED BY A CALL TO BKOPEN
 890 REM
 900 REM THE THIRD PARAMETER INDICATES THAT LOCKING IS
 910 REM TO TAKE PLACE UNCONDITIONALLY
 920 REM
 930 CALL BKLOCK(F,S$,0)
 940 REM
 950 REM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
 960 REM
 970 IF S$[1;1]<>"0" THEN DO
 980 REM N$ CONTAINS THE NAME OF THE KSAM FILE
 990 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
1000 PRINT "UNABLE TO LOCK ";$N;" ERROR ";N$;" "LS$[1;1];" DETAIL ";S$[2]
1010 CALL BKERROR(S$,M$)
1020 PRINT M$
1030 DOEND
```

Figure B-3. Dynamically Locking a KSAM File with BKLOCK

BKOPEN

A call to procedure BKOPEN initiates KSAM file processing.

```
CALL BKOPEN (filenum,status,name [,access[,lock[,exclusive[,sequence]]]])
```

In order to process a KSAM file, it must be opened with a call to the BKOPEN procedure. BKOPEN initiates processing, and optionally specifies how the file is to be processed. BKOPEN does not create the file; it must have been created previously.

To open a file means to make it available for processing. You can also specify how the file is to be accessed (whether for input, output, input/output, or for update), whether dynamic locking is allowed, whether access to the file can be shared, and whether records written to the file are to be checked for primary key sequence. Default values are assigned for the optional parameters. If you want to change the current processing or access method, you must close the file and then open it again with the parameters set to new values.

Parameters

filenum — A numeric variable whose value identifies the file opened by the call to BKOPEN. Since the value of *filenum* identifies the file in other CALL statements, it must not be changed while the file is open. (*Required parameter*)

status — A four-character string variable to which is returned a code to indicate whether or not the file was successfully opened and if not, why not. The first character is 0 if the open is successful, to another value if not. (*Required parameter*)

name — A string expression containing the name of the KSAM file to be processed. This name is the actual designator assigned to the file when it was created, or else it is a back reference to a formal designator specified in a FILE command, in which case, *name* has the form **formal designator*. (*Required parameter*)

access A numeric expression whose value indicates one of the permissible access types:

- 0 *Read only*. Use of procedures `BKWRITE`, `BKREWRITE`, and `BKDELETE` are prohibited.
- 1 *Write only*. Overwrites previously written data. Use of the procedures `BKREAD`, `BKREADBYKEY`, `BKREWRITE`, `BKDELETE`, and `BKSTART` are prohibited.
- 2 *Write only*. Saves previously written data and adds data. Use of the procedures `BKREAD`, `BKREADBYKEY`, `BKREWRITE`, `BKDELETE`, and `BKSTART` are prohibited.
- 3 *Read and write*. Use of procedures `BKREWRITE` and `BKDELETE` prohibited. (*Default value.*)
- 4 *Update access*. Allows all procedures described in this section.

(*Optional parameter*) Default: If omitted or out of range, access is 3, read and write access.

lock A numeric expression whose value indicates whether dynamic locking can take place. Acceptable values are:

- 0 Disallow dynamic locking and unlocking. Use of procedures `BKLOCK` and `BKUNLOCK` prohibited. (*Default value.*)
- 1 Allow dynamic locking and unlocking. Procedures `BKLOCK` and `BKUNLOCK` may be used to permit or restrict concurrent access to the file.

(*Optional parameter*) Default: If omitted or out of range, lock equals 0 to disallow dynamic locking.

exclusive A numeric expression whose value indicates the kind of exclusive access desired for this file. If this parameter is omitted or is not one of the following acceptable values, the default is assumed:

- 0 Depends on access parameter. If *access* = 0 (read only), then users share access to this file as if *exclusive* were set to 3. If *access* is not = 0, then access to this file is exclusive as if *exclusive* were set to 1.
- 1 Exclusive. Prohibits other access to this file until either the file has been closed or the process terminated. Only the user who opened the file can access it while it is currently open.

2 Semi-exclusive. Other users can access this file, but only for read access. The file cannot be accessed to write, rewrite, or delete records until it is closed or the process is terminated. (Default value.)

3 Shared. Once the file is opened, it can be accessed concurrently by any user in any access mode, subject only to the MPE security provisions in effect.

(*Optional parameter*) Default: If omitted or out of range, exclusive equals 2, semi-exclusive access.

sequence A numeric expression whose value indicates whether records written to the file will be checked for primary key sequence or not. Acceptable values are:

0 No sequence checking. When records are written to the file, primary key values can be in any order; their sequence is not checked. (*Default value.*)

1 Sequence checking. As each record is written to the file, KSAM checks to ensure that its primary key value is greater than the primary key value of any previously written records. If duplicates are allowed for this key, then the primary key can be equal to that of the previously written record.

(*Optional parameter*) Default: If omitted or out of range, sequence = 0, no sequence checking.

Operation Notes

After calling `BKOPEN`, you should always check the *status* parameter to determine whether the open was successful. Upon successful execution of `BKOPEN`, the file named in *name* is available for processing. An identification number is assigned to this file and returned to *filenum* where it is available to identify the open file in other calls. Until the file is successfully opened with `BKOPEN`, no operation can be executed that references the file either explicitly or implicitly.

If only the first three parameters are specified and the file is opened successfully, the file has the following default characteristics:

- Read and write access: you can read from and write to but not update the file.
- Semi-exclusive access: other users can read from but not write to or update the file.
- Dynamic locking not allowed: you cannot lock or unlock a file.
- No sequence checking: records can be written in any order without checking sequence of primary key values.

There are two types of write only access. One clears any existing records before writing the specified records to the file (*access* = 1). The other saves existing records and writes the new records after those already written (*access* = 2). Both these access modes do not permit any read or update access to the file.

Read-only access (*access* = 0) can be specified if you want to ensure that the file is not changed. This mode prohibits the writing of new records, and rewriting or deleting of existing records. In read-only mode, you can position the file and read records in either sequential or random order.

The default access mode (*access* = 3) allows you both to read records from and write records to a file, but not to change or delete existing records. If you plan to read and write records during the same process but do not want to alter existing records, use this access mode.

If you want to rewrite or delete existing records in a KSAM file, you must open with *access* = 4. This mode allows you to use the `BKREWRITE` and `BKDELETE` procedures, as well as all the other procedures described in this section.

Table B-2 summarizes the procedures you may call depending on the *access* parameter value you specify in `BKOPEN`.

Table B-2.
Procedures Allowed by BKOPEN Access Parameter

Procedure	Read-only (*access*=0)	Write-only with Clear (*access*=1)	Write-only with Save (*access*=2)	Read/Write (*access*=3)	Update (*access*=4)
`BKREAD`	X			X	X
`BKREADBYKEY`	X			X	X
`BKSTART`	X			X	X
`BKWRITE`		X	X	X	X
`BKREWRITE`					X
`BKDELETE`					X
`BKCLOSE`	X	X	X	X	X
`BKERROR`	X	X	X	X	X

By default in a multi-user environment, all users whose MPE security restrictions allow them to access your file can read the file, but they cannot change the file or add new records to it. This is the default specification of the *exclusive* parameter in `BKOPEN` (*exclusive*=2). It is independent of the value of the *access* parameter.

If you want to prevent other users from reading the file as well as writing to it, you must specify this by setting *exclusive*=1. This setting allows only you to read from, write to, or alter the file.

Another alternative is to set *exclusive*=0, thereby allowing other users access to the file only when it is opened for read only (*access*=0). This setting of the *exclusive* parameter prevents any access by other users when the file is opened for any form of write or update (*accesss* $\neq$ 0). This means that you and other users share read access to the file, but only you can write to or change the file.

You can choose to completely share access to the file, reading and/or writing and updating, by setting the *exclusive* parameter to 3.

(Refer to Table B-3 for a summary of the relation between the *exclusive* parameter and the *access* parameter.)

Table B-3.
Relationship of Exclusive Parameter to Access Parameter

	exclusive=**0**	*exclusive*=**1**	*exclusive*=**2** **(default)**	*exclusive*=**3**
access=0 (read only)	shared	exclusive	semi-exclusive	shared
access$\neq$0 (write only, read/write, or update)	exclusive	exclusive	semi-exclusive	shared

When access is shared, it is good practice to allow dynamic locking so that individual users can dynamically lock the file while performing any updates to the file. The file can be unlocked as soon as the update is complete. An update to a file is when you write a new record, delete a record, or rewrite an existing record. When access is exclusive or semi-exclusive, there is no need for dynamic locking since only the user who has opened the file can update the file.

Dynamic locking should also be allowed if access is shared and you plan to read the file sequentially. This is because the sequential read procedure (`BKREAD`) is dependent on the position of the logical record pointer and, in a shared environment, this pointer can be changed by other users unless the file is locked. (Refer to Table B-1 for a list of the pointer-dependent procedures.)

When sequence checking is specified, you must write records to the file in primary key sequence. An attempt to write a record out of sequence causes the write to fail and the value 21 is returned to *status* following a call to `BKWRITE`. As a result of sequence checking, the physical and the primary key sequence of records in your file is the same. Since the BASIC KSAM procedures have no provision to

read the file in physical sequence, you may want to specify sequence checking for any file that you will want to read in that order. With sequence checking, a file read in logical order by primary key (the default for `BKREAD`) is also read in physical order.

The example in Figure B-4 shows how to use `BKOPEN` to open a KSAM file for input and output (default *access*), with dynamic locking (*lock*=1), for shared access (*exclusive*=3), and without sequence checking (default *sequence*).

```
 10 DIM S$[4] <-------- status \
 20 DIM N$[26] <------------ filename |- variable dimensions
 30 DIM M$[72] <-------- message /
 40 INTEGER A[10]
 50 DIM B$[12]
 55 INTEGER J
 60 DIM B1$[1]
 65 DIM B2$[2]
 70 INTEGER A2[2],A3[3],A5[5]
 80 REM
 90 REM THE KSAM/3000 FILE WAS BUILT WITH:
100 REM REC=-80,16,F,ASCII
110 REM KEY=B,2,2,,DUP
120 REM SO,RECORD LENGTH IS 80 BYTES, FIXED, TYPE ASCII, 16 REC/BLOCK.
130 REM THE KEY IS 2 CHARACTERS LONG,STARTING IN CHARACTER 2 OF RECORD
135 REM
140 REM ******************************************************
145 REM * OPEN A KSAM FILE *
150 REM ******************************************************
160 REM
170 REM THE FILE NAME IS IN N$
175 REM THE STATUS OF THE CALL IS RETURNED IN S$
180 REM WHEN SUCCESSFUL, BKOPEN RETURNS A FILE NUMBER IN F
190 REM INPUT-OUTPUT ACCESS IS SPECIFIED IN J
200 REM DYNAMIC LOCKING IS ALLOWED IN D
210 REM SEMI-EXCLUSIVE ACCESS IS INDICATED IN E
220 REM
240 N$="KNAME,ACCOUNT,GROUP" <---------- file name
250 J=3 <-------- access is read/write
260 D=1 <------------------------------ dynamic locking allowed
270 E=3 <-------- access shared
280 CALL BKOPEN(F,S$,N$,J,D,E)
290 REM
300 REM NOW DETERMINE WHETHER THE CALL SUCCEEDED:
310 REM
320 IF S$[1;1]<>"0" THEN DO
330 REM S$ IS THE STATUS CODE SET BY THE CALL TO BKOPEN
340 REM N$ IS THE NAME OF THE FILE
350 PRINT "UNABLE TO OPEN ";N$;" ERROR ";S$[1;1];"DETAIL "LS$[2]
360 CALL BKERROR(S$,M$)
370 PRINT M$
380 GOTO 3620 <-------- to close the file
390 DOEND
400 REM
410 REM THE PROGRAM CONTINUES
```

Figure B-4. Opening KSAM File with BKOPEN

BKREAD

Transfers the next logical record from a KSAM file to a BASIC program.

```
CALL BKREAD(filenum,status[,parameterlist])
```

A call to BKREAD transfers the contents of a record from a KSAM file to a storage area defined by a list of variables in a BASIC program. The record read is that at which the logical record pointer is currently positioned. In a series of calls to BKREAD, records are read in ascending order by key value. The primary key is used unless a previous call to BKSTART or BKREADBYKEY has positioned the pointer to an alternate key. The file must have been opened with an access mode that allows reading.

Parameters

filenum — A numeric variable containing the file number that identifies the file. This number was returned by the last call to BKOPEN. It should not be altered unless the file is closed by a successful call to BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKREAD was successful and if not, why not. The first character is set to zero when the call succeeds, to another value if not. (*Required parameter*)

parameterlist — A list of variables separated by commas into which the data in the record is read. The contents of the record are read into the variable (or variables) until the physical length (or combined physical lengths) of *parameterlist* is exhausted, or the end of the record is reached. (*Optional parameter*) Default: If omitted, the logical record pointer is positioned to the beginning of the next record in key sequence.

Operation Notes

After calling BKREAD, you should always check the *status* parameter to determine whether the read was successful. Upon successful completion of BKREAD, the variables specified in *parameterlist* contain data read from the record at which the record pointer was positioned when BKREAD was called. Note that if *parameterlist* is omitted, the record pointer is positioned to the beginning of the next logical record, effectively skipping the current record.

In order to use BKREAD, the file must be opened for input. The BKOPEN *access* parameter should be zero if you plan to only read or position a record. To both read from and write to the same open file, you either omit the *access* parameter or set it to 3. If you want to rewrite or update as well as read records, you must set *access* to 4.

Values are read from the current record into the variables specified in *parameterlist* according to the type and length of the variable. For example, consider the following code:

```
10 DIM G$(3),H$(3),S$(4)
20 INTEGER L,F
30 CALL BKREAD (F,S$,G$,H$,L)
```

If the record being read contains only the word SCRABBLE, this word is read into the specified variables as if they were assigned by the statements:

```
100 G$="SCR"
110 H$="ABB"
120 L=NUM("LE")
```

Note

Each variable in the *parameterlist* is filled to its current physical length before proceeding to the next variable.

The following calls omit the *parameterlist* in order to skip forward two records:

```
210 CALL BKREAD(F,S$)
220 CALL BKREAD(F,S$)
```

The records skipped are not the next records physically placed on the file, but are the next two in logical sequence according to the value of the current key. The particular key used for the read sequence can be selected with a call to `BKSTART` or `BKREADBYKEY`. `BKSTART` can also be used to position the file to the beginning of the record with the lowest key value in the selected key.

The example in Figure B-5 assumes that the record pointer has been positioned to the beginning of the first record in primary key sequence. Assume that the file being read was opened in the example in Figure B-4, the records read were written in the example in Figure B-11.

Each record contains five integers followed by five undefined words followed by a string of three characters. The record is read into:

A5 a 5-word integer array

A2 a 2-word integer array

A3 a 3-word integer array

B1$ a 1-character string

B2$ a 2-character string

The five integers that were written to the beginning of each record are read into array A5. The next two arrays A2 and A3 receive the undefined values that filled the next five words of the record. The first string character is read into B1$, the next two into B2$.

If you open the file for read-only access (*access*=0), and the *exclusive* parameter is allowed to default to zero, then more than one user can share read access to the file. In this case, or if you specifically indicate shared access, you should also allow dynamic locking in order to read records from the file in key sequence. This is necessary because `BKREAD` depends on the current position of the logical record pointer. (Refer to Table B-1 for a list of the pointer-dependent procedures.)

For example, if you plan to read the file sequentially starting from a particular key value, use the following sequence of calls:

```
BKOPEN <-------- open file for read-only, shared access, allow dynamic locking
BKLOCK <-------- lock file
BKSTART <-------- position pointer
BKREAD loop <-------- read file in sequence from original pointer position
BKUNLOCK <-------- unlock file when last record read
```

```
   10 DIM S$[4]
   20 DIM N$[26]
   30 DIM M$[72]
   40 INTEGER A[10]
   50 DIM B$[12]
   55 INTEGER J
   60 DIM B1$[1]
   65 DIM B2$[2]
   70 INTEGER A2[2],A3[3],A5[5]
    :
 1310 REM ****************************************************
 1320 REM * READ FROM A KSAM FILE * o
 1330 REM ****************************************************
 1350 REM F IS THE FILE NUMBER OF A KSAM FILE
 1360 REM OPENED BY A CALL TO BKOPEN
 1370 REM
 1380 REM AN ASSUMPTION HAS BEEN MADE THAT THE RECORD TO BE READ
 1390 REM CONTAINS THE SAME INFORMATION THAT WAS WRITTEN TO
 1400 REM THE FILE BY THE EXAMPLE TO WRITE A KSAM FILE
 1410 REM
 1420 CALL BKREAD(F,S$,B1$,B2$,A5[*],A3[*],A2[*])
 1430 REM
 1440 REM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
 1450 REM
 1460 IF S$[1;1]<>"0" THEN DO
 1470 REM N$ CONTAINS THE NAME OF THE KSAM FILE
 1480 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
 1490 PRINT "UNABLE TO READ ";N$;" ERROR ";S$[1;1];" DETAIL ";S$[2]
 1500 CALL BKERROR(S$,M$)
 1510 PRINT M$
 1520 REM
 1530 REM TEST FOR END OF FILE
 1540 REM AND POSITION TO LEAST VALUED PRIMARY KEY
 1550 IF S$[1;1]="1" THEN 1080
 1560 GOTO 3620
 1570 DOEND
 1580 REM
 1590 REM ECHO WHAT WAS READ
 1600 REM
 1610 PRINT "RECORD CONTAINS";B1$,B2$
 1620 MAT PRINT A5
 1622 MAT PRINT A3,A2
 1630 REM
 1650 REM THE CONTENTS OF B1$="1", OF B2$="23"
 1660 REM THE CONTENTS OF A5(1) THROUGH A5(5) ARE 1 THROUGH 5.
 1670 REM THE CONTENTS OF A3 AND A2 ARE UNKNOWN.
 1680 REM
 1690 REM THE PROGRAM CONTINUES
```

Figure B-5. Reading From a KSAM File with BKREAD

BKREADBYKEY

Transfers record identified by particular key value from KSAM file to BASIC program.

`CALL BKREADBYKEY(`*filenum,status,keyvalue,keylocation,parameterlist*`)`

A call to `BKREADBYKEY` locates and reads a record into a storage area identified by a list of variables in the BASIC program. The record to be read is located by matching the specified *keyvalue* with an identical value stored in the record starting at *keylocation*. The record value and the value specified in *keyvalue* must match exactly, or an error code is returned to *status*. To use `BKREADBYKEY`, the file must be open in an access mode that allows reading.

You cannot use `BKREADBYKEY` to locate a record by generic or approximate key values. For this purpose you can call `BKSTART` followed by a call to `BKREAD`.

Parameters

filenum — A numeric variable containing the file number that identifies the file. This number was returned by the last call to `BKOPEN`. It should not be altered unless the file is closed with a successful call to `BKCLOSE`. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to `BKREADBYKEY` was successful and if not, why not. The first character is set to zero if the call succeeds, to another value if not. (*Required parameter*)

keyvalue — A string or numeric expression whose value is compared to a key value in the record. The record pointer is positioned to the first record with a key value at *keylocation* that is exactly equal to the specified *keyvalue*. In order to match exactly, the record value and *keyvalue* must have the same logical length. (*Required parameter*)

keylocation — A numeric expression whose value indicates the starting character position in each record of the key used to locate the record to be read by `BKREADBYKEY`. The characters in a record are counted starting with 1. If the value of *keylocation* is zero, the primary key is assumed. The primary key also may be specifically indicated by its location in the record. (*Required parameter*)

parameterlist — A list of variables separated by commas into which the data in the record is read. The contents of the record are read into the variable (or variables) until the physical length (or combined physical lengths) of *parameterlist* is exhausted, or until the end of the record is reached. (*Required parameter*)

Operation Notes

After calling BKREADBYKEY, you should always check the *status* parameter to determine whether the read was successful. Upon completion of BKREADBYKEY, the variables specified in *parameterlist* contain data read from the record located through the *keyvalue* and *keylocation* parameters.

The key value in the record to be read must exactly match the specified *keyvalue*. Unlike BKSTART, the only relation between the value in the record and the value in the call is that of equality. If duplicate key values are allowed in the key being sought, then the first record with a matching key value is read by BKREADBYKEY. To read the remaining records with duplicate key values, you should use BKREAD.

Note

Each variable in *parameterlist* is filled to its current physical length before proceeding to the next variable.

The example in Figure B-6 uses BKREADBYKEY to read the first record found with the value 23 starting in byte 2. Since this is the file written by BKWRITE in Figure B-11, the records in the file are identical including the keys and only the first record is read.

```
2220 REM ****************************************************
2230 REM * READ BY KEY FROM A KSAM FILE *
2240 REM ****************************************************
2250 REM
2260 REM F IS THE FILE NUMBER OF A KSAM FILE
2270 REM OPENED BY A CALL TO BKOPEN
2280 REM
2290 REM AN ASSUMPTION HAS BEEN MADE THAT THE RECORD TO BE READ
2300 REM CONTAINS THE SAME INFORMATION THAT WAS WRITTEN IN THE
2310 REM WRITE EXAMPLE.
2320 REM
2330 REM AN ADDITIONAL ASSUMPTION IS THAT THE DESIRED KEY VALUE
2340 REM STARTS AT CHARACTER 2 AND HAS THE VALUE "23".
2350 REM
2360 CALL BKREADBYKEY(F,S$,"23",2,B1$,B2$,A5[*],A3[*],A2[*])
2370 REM
2380 REM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
2390 REM
2400 IF S$[1;1]<>"0" THEN DO
2410 REM N$ CONTAINS THE NAME OF THE KSAM FILE
2420 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
2430 PRINT "UNABLE TO READBYKEY ";N$;" ERROR ";S$[1;1];" DETAIL "S$[2]
2440 CALL BKERROR(S$,M$)
2450 PRINT M$
2460 GOTO 3620
2470 DOEND
2480 REM
2490 REM THE CONTENTS OF B1$="1", OF B2$="23".
2500 REM THE CONTENTS OF A5(1) THROUGH A5(5) ARE INTEGERS 1 THROUGH 5
2510 REM THE CONTENTS OF A3 AND A2 ARE UNKNOWN.
2520 REM
2530 REM ECHO WHAT WAS READ
2540 REM
2550 PRINT "RECORD READ = ";B1$,B2$
2560 MAT PRINT A5
2562 MAT PRINT A3,A2
2570 REM
2580 REM THE PROGRAM CONTINUES
```

Figure B-6. Reading a Record Located by Key Value with BKREADBYKEY

BKREWRITE

Changes the contents of a record in a KSAM file.

CALL BKREWRITE (*filenum, status, parameterlist*)

A call to BKREWRITE replaces the contents of an existing record with new values. The record to be rewritten is the last record accessed by a call to BKREAD, BKREADBYKEY, or BKSTART. To use BKREWRITE, the file must be open in the access mode that allows update. If access is shared, it must also be opened with dynamic locking allowed, and the file must be locked by BKLOCK before records are rewritten.

Parameters

filenum — A numeric variable containing the file number that identifies the file. This number was returned by the last call to BKOPEN. It should not be altered unless the file is closed with a successful call to BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKREWRITE was successful and if not, why not. The first character is set to zero if the call succeeds, to another value if not. (*Required parameter*)

parameterlist — A list of variables or constants, separated by commas, that contains the data to be written to the file replacing the last record read or written. The total length of the new record is derived from the total number, data type, and length in characters of each item in *parameterlist*. Although this length need not be the same as the record it replaces, it should be long enough to contain all the keys, but not exceed the defined record length. (*Required parameter*)

Operation Notes

After calling BKREWRITE, you should always check the *status* parameter to make sure that the rewrite was successful. Upon successful completion of BKREWRITE, new values replace the data in the last record read to or written from the BASIC program. The new data may change every value in the previously read record including the primary key value.

If you want to replace a record with a particular key value, you should locate and read the record with BKREADBYKEY or BKSTART. To rewrite a series of records you should read the records with BKREAD.

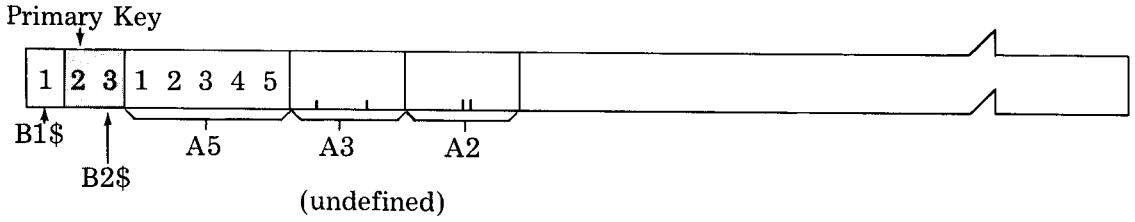
When the data in the *parameterlist* of BKREWRITE is shorter in total length than the data in the record being rewritten, there is less total data in the rewritten record. In order to maintain the key sequence of all keys, defined values should be written to the location of all keys, both the primary key and any alternate keys.

Note

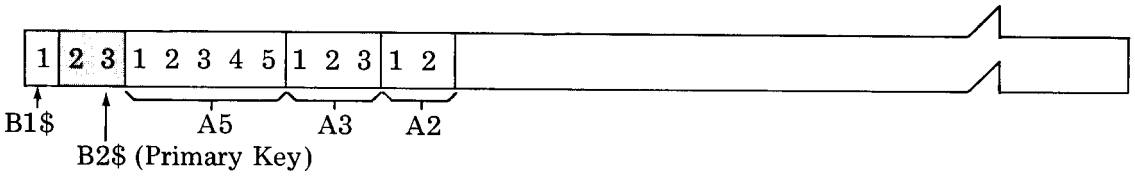
Items written to a KSAM file with the BKREWRITE procedure are concatenated; rounding to halfword boundaries does not occur.

The example in Figure B-7 writes new values to a record originally written in Figure B-11 and read in Figure B-5. The new values fill an array that had undefined values in the last five elements, now defined as two arrays A3 and A2 by the *BKREAD* call. The primary key value 23 in location 2 is unchanged.

The record read by *BKREAD* contained the following values:



After being rewritten by BKREWRITE, it contains the following values:



When access is shared, the call to BKREAD, BKREADBYKEY, or BKSTART that locates the record to be rewritten should be included in the same pair of BKLOCK/BKUNLOCK calls as the call to BKREWRITE. This ensures that no other user alters the record pointer between the call that locates the record and the call that rewrites it.

If you want to sequentially rewrite all records in a chain of records with duplicate keys, locate the first record in the chain with BKREADBYKEY. Then call BKREWRITE to modify this record. If no key value (the selected key or any other) is modified, subsequent calls to BKREWRITE will modify the next sequential records in the chain of duplicate keys. If, however, any key has been changed, the modified key is written to the end of the chain and the next sequential record is one with the next higher key value. In this case, to rewrite all records with duplicate keys, precede each call to BKREWRITE by a call to BKREADBYKEY.

```
2600 REM
2610 REM ************************************************************
2620 REM * REVISE THE CONTENTS OF A RECORD READ FROM A KSAM FILE *
2630 REM ************************************************************
2640 REM
2650 REM F IS THE FILE NUMBER OF A KSAM FILE OPENED BY A CALL TO BKOPEN
2660 REM NOTE THAT FOR BKREWRITE,BKOPEN ACCESS MODE MUST=4 FOR UPDATE.
2670 REM
2680 REM AN ASSUMPTION HAS BEEN MADE THAT THE RECORD TO BE READ
2690 REM CONTAINS THE SAME INFORMATION THAT WAS WRITTEN TO THE
2700 REM KSAM FILE IN THE BKWRITE EXAMPLE,,
                |------------------ parameterlist
2710 REM /------------------------\
2720 CALL BKREAD(F,S$,B1$,B2$,A5[*],A3[*],A2[*])
2730 REM
2740 REM NOW DETERMINE WHETHER THE CALL HAS SUCCEEDED.
2750 REM
2760 IF S$[1;1]<>"0" THEN DO
2770 REM N$ CONTAINS THE NAME OF THE KSAM FILE
2780 REM S$ CONTAINS THE STATUS CALL SET BY THE PRECEDING CALL
2790 PRINT "UNABLE TO READ ";N$;" ERROR ";S$[1;1]" DETAIL ";S$[2]
2800 CALL BKERROR(S$,M$)
2810 PRINT M$
2820 GOTO 3620
2830 DOEND
2900 REM THE CONTENTS OF B1=1", OF B2$="23"
2910 REM THE CONTENTS OF A5(1) THROUGH A5(5) ARE 1 THROUGH 5
2920 REM THE CONTENTS OF A3 AND A2 ARE UNKNOWN
2930 REM
2940 REM STORE VALUES 1 THROUGH 3 INTO A3(1) THROUGH A3(3)
2950 REM STORE VALUES 1 AND 2 INTO A2(1) AND A2(2).
2960 REM
2970 FOR I=1 TO 2
2980 A2[I]=I
2990 A3[I]=I
3000 NEXT I parameterlist
3010 A3[3]=3 |
3020 REM /------------------------\
3030 CALL BKREWRITE(F,S$,B1$,B2$,A5[*],A3[*],A2[*])
3040 REM
3050 REM NOW DETERMINE WHETHER THE CALL HAS SUCCEEDED
3060 REM
```

Figure B-7. Rewriting Record in KSAM File with BKREWRITE

```
3070 IF S$[1;1]<>"0 THEN DO
3080 REM N$ CONTAINS THE NAME OF THE KSAM FILE
3090 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
3100 PRINT "UNABLE TO REWRITE ";N$;" ERROR ";S$[1;1];" DETAIL ";S$[2]
3110 CALL BKERROR(S$,M$)
3120 PRINT M$
3130 GOTO 3620
3140 DOEND
3150 REM
3160 REM ECHO WHAT WAS UPDATED
3170 REM
3180 PRINT "REWRITTEN RECORD = ";B1;B2
3190 MAT PRINT A5,A3,A2
3200 REM
3210 REM THE PROGRAM CONTINUES
```

Rewriting Record in KSAM File with BKREWRITE (continued)

BKSTART

Positions a KSAM file to a particular record based on a key value.

`CALL BKSTART(`*filenum,status*`[,`*keyvalue*`[,`*keylocation* `[,`*relation* `]]])`

By calling `BKSTART`, you can position the record pointer to any record in the file based on the value of a key in that record. The key can be the primary key or any alternate key, since `BKSTART` also allows you to select the key for positioning and for subsequent sequential reads. If you want to read all the keys in a key sequence, you can use BKSTART to position the pointer to the record with the lowest key value in the selected key.

Parameters

filenum — A numeric variable containing the file number that identifies the file. This number was returned by the last call to `BKOPEN`. It should not be altered unless the file is closed with a successful call to `BKCLOSE`. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to `BKSTART` was successful and if not, why not. The first character is set to zero when the call succeeds, to another value when it fails. (*Required parameter*)

keyvalue — A string or numeric expression whose value is compared to a key value in this record. The record pointer is positioned to the first record with a key value that bears the relation specified by *relation* to the value in *keyvalue*. If the value is a string, its logical length is used for the comparison; otherwise, the physical or dimensioned length is used. The length of this value must be less than or equal to the length of the key as specified when the file was created. If *keyvalue* is a null string (""), the file is positioned to the beginning of the first logical record according to the value of the key in *keylocation*. (*Optional Parameter*)

Default: If omitted, the value assumed for keyvalue is the lowest value for the specified key type.

keylocation — A numeric expression whose value indicates the starting character location in each record of the key used for positioning by `BKSTART`. The characters in a record are counted starting with one. If set to zero, the primary key is assumed. (*Optional parameter*)

Default: If omitted, the primary key is assumed.

relation A numeric expression whose value specifies the relation between the specified *keyvalue* and the value of the key at *keylocation*. The record pointer is positioned to the first record with a key value satisfying this relation:

0	The value of the record key is equal to *keyvalue*
1	The value of the record key is greater than *keyvalue*
2	The value of the record key is greater than or equal to *keyvalue* (*default*).
> 2	Any value greater than 2 is treated as if it were 2. (*Optional parameter*)
Default	If omitted, the relation is assumed to be 2, *record key* is greater than or equal to the *keyvalue*.

Operation Notes

After calling `BKSTART`, you should check the status parameter to determine if the procedure was executed successfully. If successful, the record pointer is positioned at the beginning of the first record with a value at *keylocation* that has the relation specified in *relation* to the value specified in *keyvalue*.

If default values are assumed for all three optional parameters, the pointer is positioned to the record with the lowest value for its type in the primary key location.

If the relation specified is equality (*relation* = 0), then a record must be located that has the same key value as that specified in the `BKSTART` call. When the record is found, the pointer is positioned to it. If duplicate values are allowed for the key, then the pointer is positioned at the first record with the particular key value.

When the specified relation is greater than (*relation* = 1), the file is searched until a record is found with a key value greater than the specified key value. The search passes over any record with a key value equal to the specified value. This relation allows you to retrieve items by an *approximate key*. Thus, if you specify a key value of "R", a call to `BKSTART` will position the pointer to the first record with a key value that starts with the letter R. A subsequent series of calls to `BKREAD` allows you to read the remaining records in the file or, by including a test, to read only the records beginning with R.

When the specified relation is greater than or equal to (*relation* = 2), `BKSTART` looks for a record containing a value equal to the specified value. If found, it positions the pointer to that record. If not found, it continues looking and positions the pointer to the first record that is greater than the specified value. This type of search can be used to locate records by *generic key*. A generic, or partial, key is a value that matches characters at the beginning of the key, but not necessarily the end.

Whenever a record cannot be found with a key that satisfies the relation and value specified, the value 23 for invalid key is returned to *status*.

BKSTART allows you to specify a key other than the primary key assumed by BKREAD. Called prior to a series of calls to BKREAD, it prepares for a sequential read of the file in alternate key order. For example, assuming a file with an alternate key in location 21, the following call positions the pointer to the first record in that key sequence:

```
100 DIM A$(10),S$(4)
150 A$=" " <------------------- assign null string to keyvalue
160 L=21 <-------------------- assign alternate key location to keylocation
170 CALL BKSTART(F,S$,A$,21)
```

The default for *relation* is 2 (greater than or equal to) and need not be specified except for documentation purposes.

Figure B-8 illustrates the use of BKSTART with default values for all optional parameters. Specified in this minimal form, it positions to the least valued primary key.

```
1080 REM ******************************************************
1090 REM * POSITION TO LEAST VALUED PRIMARY KEY *
1100 REM ******************************************************
1110 REM
1120 REM F IS THE FILE NUMBER OF A KSAM FILE
1130 REM OPENED BY A CALL TO BKOPEN
1140 REM
1150 CALL BKSTART(F,S$)
1160 REM
1170 REM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
1180 REM
1190 IF S$[1;1]<>"0" THEN DO
1200 REM N$ CONTAINS THE NAME OF THE KSAM FILE
1210 REM S$ CONTAINS THE STATUS CODE RETURNED BY THE PRECEDING CALL
1220 PRINT "UNABLE TO POSITION FILE TO LEAST VALUED PRIMARY KEY"
1230 PRINT "ERROR ";S$[1;1]," DETAIL";S$[2]
1240 CALL BKERROR,(S$,M$)
1250 PRINT M$
1260 GOTO 3620
1270 DOEND
1280 REM
1290 REM THE PROGRAM CONTINUES
1300 REM
```

Figure B-8. Positioning Pointer to Least-Valued Record with BKSTART

The example in Figure B-9 positions the record pointer to a record containing a specific key value. The value is 23; it is located starting in the second character of each record. The value for *relation* is zero indicating that the key must contain exactly the value 23, not a value larger than 23.

```
1920 REM
1930 REM ************************************
1940 REM * POSITION A KSAM FILE *
1950 REM ************************************
1960 REM
1970 REM F IS THE FILE NUMBER OF A KSAM FILE
1989 REM OPENED BY A CALL TO BKOPEN
1990 REM
2000 REM AN ASSUMPTION HAS BEEN MADE THAT THE POSITIONING TO BE
2010 REM DONE IS TO THE RECORD WRITTEN IN THE WRITE EXAMPLE,
2020 REM AND THAT THE DESIRED KEY STARTS AT CHARACTER 2.
2060 REM
2070 CALL BKSTART(F,S$,"23",2,0)
2080 REM
2090 REM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
2100 REM
2110 IF S$[1;1]<>"0" THEN DO
2120 REM N$ CONTAINS THE NAME OF THE KSAM FILE
2130 REM S$ CONTAINS THE STATUS CODE RETURNED BY THE PRECEDING CALL
2140 PRINT "UNABLE TO START ";N$;" ERROR ";S$[1;1];" DETAIL ";S$[2]
2150 CALL BKERROR(S$,M$)
2160 PRINT M$
2170 GOTO 3620
2180 DOEND
2190 REM
2200 REM THE PROGRAM CONTINUES
2210 REM
```

Figure B-9. Positioning Pointer to Particular Record with BKSTART

BKUNLOCK

Unlocks a KSAM file dynamically locked by BKLOCK.

CALL BKUNLOCK(*filenum,status*)

A file locked by BKLOCK is released for use by other users with a call to BKUNLOCK. (If you log off from any connection with the system, the file is also unlocked.) Since dynamic locking takes place during shared access to the same file by more than one user, it is important that any file locked by BKLOCK be unlocked as soon as possible by BKUNLOCK.

To use BKUNLOCK, the file must be opened with dynamic locking allowed by all users who share access to the file.

Parameters

filenum — A numeric variable containing the file number that identifies the file. This number was returned to *filenum* by the last call to BKOPEN. It should not be altered until the file is successfully closed by BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKLOCK was successful and if not, why not. The first character is set to zero when the call succeeds, to another value if it fails. (*Required parameter*)

Operation Notes

After calling BKUNLOCK, you should always check the status parameter to make sure that the procedure was successfully executed. When successful, a file locked by BKLOCK is again made available for access by other users. If the file is not locked by BKLOCK when BKUNLOCK is called, the file is not affected.

Figure B-10 illustrates the use of BKUNLOCK to unlock the file after it is updated.

```
1700 REM ****************************************************
1710 REM * UNLOCK A KSAM FILE *
1720 REM ****************************************************
1730 REM
1740 REM F IS THE FILE NUMBER OF A KSAM FILE
1750 REM OPENED BY A CALL TO BKOPEN
1760 REM
1770 CALL BKUNLOCK(F,S$)
1780 REM
1790 REM NOW DETERMINE WHETHER THE CALL HAS SUCCEEDED
1800 REM
1810 IF S$(1;1)<>"0" THEN DO
1820 REM N$ CONTAINS THE NAME OF THE KSAM FILE
1830 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CALL
1840 PRINT "UNABLE TO UNLOCK ";N$;" ERROR ";S$(1;1);"DETAIL ";S$[2]
1850 CALL BKERROR(S$,M$)
1860 PRINT M$
1870 GOTO 3620
1880 DOEND
1890 REM
1900 REM THE PROGRAM CONTINUES
```

Figure B-10. Dynamically Unlocking a KSAM File

BKWRITE

Writes data from a BASIC program to a KSAM file.

```
CALL BKWRITE (filenum,status,parameterlist)
```

A call to procedure BKWRITE writes a record to a KSAM file from a BASIC program. This call provides the only way to create a KSAM record from a BASIC program. The file must have been opened with an access mode that allows writing. If access is shared, the file also must be opened for dynamic locking (*lock* = 1), and the file locked with BKLOCK before any records are written.

Parameters

filenum — A numeric variable containing the file number value that identifies the file. This number was returned by the last call to BKOPEN. It should not be altered unless the file is closed by a successful call to BKCLOSE. (*Required parameter*)

status — A four-character string variable to which is returned a code that indicates whether or not the call to BKWRITE was successful and if not, why not. The first character is set to zero when the call succeeds, to another value if not. (*Required parameter*)

parameterlist — A list of variables or constants, separated by commas, that contain the data to be written to the file as a record. The total length of the record contents is derived from the total number, the type, and the length in characters of the items in *parameterlist*. The *parameterlist* must contain a value for each location defined as a key location in the record. (*Required parameter*)

Operation Notes

After calling BKWRITE, you should always check the *status* parameter to ensure that the write was successful. Upon successful completion of BKWRITE, one record containing the values specified in *parameterlist* is written to the opened KSAM file.

Two parameters that are set when the file is opened affect how BKWRITE operates. These are the *access* and *sequence* parameters.

In order to write to a file, the file must be opened with *access* greater than 0. If the *access* parameter is set to 1, all existing data in the file is cleared before the first record is written to the file. If *access* is set to 2 or greater, the first record written by BKWRITE immediately follows any existing records; the file is not cleared.

The *sequence* parameter determines whether records must be written in primary key sequence, or not. If *sequence* is 0, records can be written in any order; no check is made on the sequence of the primary key field. If *sequence* is set to 1, you must write each record with a value in the primary key field that is greater than the primary key value in the previous record. Primary key values may equal the

previous primary key value only if the file was created with duplicate key values permitted.

Note

Items written to a KSAM file from a BASIC program are concatenated; rounding to halfword boundaries does not occur.

Figure B-11 is an example of writing one string and one integer array to each record of the KSAM file.

```
 10 DIM S$[4]
 20 DIM N$[26]
 30 DIM M$[72]
 40 INTEGER A[10]
 50 DIM B$[12]
 55 INTEGER J
 60 DIM B1$[1]
 65 DIM B2$[2]
 70 INTEGER A2[2],A3[3],A5[5]
 80 REM
 90 REM THE KSAM/3000 FILE WAS BUILT WITH:
100 REM REC=-80,16,F,ASCII
110 REM KEY=B,2,2,,DUP
120 REM S0,RECORD LENGTH IS 2 BYTES, FIXED, TYPE ASCII, 16 REC/BLOCK.
130 THE KEY IS 2 CHARACTERS LONG,STARTING IN CHARACTER 2 OF RECORD
135 REM
 :
430 REM ******************************************************
440 REM * WRITE TO A KSAM FILE *
450 REM ******************************************************
460 REM
470 REM ASSIGN VALUES TO OUTPUT VARIABLES
480 REM
490 FOR I=1 TO 5
500 A[I]=I
510 NEXT I
520 RS="123"
530 REM
540 REM F IS THE FILE NUMBER OF A KSAM FILE
550 REM OPENED BY A CALL TO BKOPEN
560 REM
```

Figure B-11. Writing to a KSAM File with BKWRITE

```
570 REM NOTE THAT ONLY THREE BYTES "123" ARE WRITTEN FROM B$
580 REM WHEREAS TEN WORDS ARE WRITTEN FROM NUMERIC ARRAY A.
620 REM
630 REM THREE IDENTICAL RECORDS ARE BEING OUTPUT SO THAT
640 REM SUBSEQUENT EXAMPLES OF THIS PROGRAM WILL EXECUTE
650 REM .
660 FOR I=1 TO 3
670 CALL BKWRITE(F,S$,BS,A[*])
680 REM
690 REM NOW DETERMINE WHETHER THIS CALL SUCCEEDED
700 REM
710 IF S$[1;1]<>"0" THEN DO
720 REM N$ CONTAINS THE NAME OF THE KSAM FILE
730 REM S$ CONTAINS THE STATUS CODE SET BY THE PRECEDING CODE
740 PRINT "UNABLE TO WRITE TO ";N$;"ERROR "[S$]; DETAIL ";S$[     2]
750 CALL BKERROR(S&,Ms)
760 PRINT M$
770 GOTO 3620
780 DOEND
790 NEXT I
800 REM
810 REM THE PROGRAM CONTINUES
```

Writing to a KSAM File with BKWRITE (continued)

C

HP C/iX Example Program

The following example program shows how a KSAM XL file can be created, accessed, and updated from an HP C/iX program. This program uses features of ANSI C. Compile with `INFO=-Aa + e`.

This example program uses the `assert` macro to do quick error checking. In a production program, more comprehensive error checking and reporting would be desirable.

The KSAM XL file has the following layout:

```
 1 - 5  Employee number        (primary key)
 6 - 25 Name                   (secondary key)
26 - 34 Social Security Number
35 - 38 Department Number      (secondary key)
39 - 44 Date of hire
```

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpe.h>
#pragma intrinsic FCLOSE, FFINDN, FLOCK
#pragma intrinsic FREAD, FREADBYKEY, FREMOVE
#pragma intrinsic FUNLOCK, FUPDATE, FWRITE
#pragma intrinsic HPCICOMMAND, HPFOPEN
#define FILENAME "KSAMD"
typedef char record_t[44];
static int filenum;
static void close_file(void);
static void create_file(void);
static void delete_records(void);
static void dump_file(void);
static void list_sequential(void);
static void list_sequential_primary(void);
static void list_sequential_secondary(int location);
static void lock_file(void);
static void open_file(void);
static void unlock_file(void);
static void update_records(void);
static void write_new_records(void);
static void write_record(const char *record);
main(void)
{
   create_file();
```

```
   open_file();
   dump_file();
   write_new_records();
   update_records();
   delete_records();
   dump_file();
   close_file();
   return EXIT_SUCCESS;
}
static void close_file(void)
{
   /* Close file */
   FCLOSE(filenum, 0, 0);
   assert(ccode()==CCE);
}
static void create_file(void)
{
   /* Create sample KSAM XL file and load initial test data */
   int status; short cmderror;
   const int ksamxl=3, out=1, recsize=sizeof(record_t),
       filesize=100, save=1, ascii=1;
   const struct
      {
      short filler_1[10];
      unsigned short language_id      : 16;
      short filler_2[4];
      struct
         {
         unsigned short filler_1      : 10;
         unsigned short chg_primary   : 1;
         unsigned short kslang        : 1;
         unsigned short ksreuse       : 1;
         unsigned short seq_random    : 1;
         unsigned short rec_numbering : 1;
         unsigned short filler_2      : 1;
         } flagword;
      unsigned short filler_3         : 8;
      unsigned short num_keys         : 8;
      struct
         {
         unsigned short key_type      : 4;
         unsigned short key_length    : 12;
         unsigned short key_location  : 16;
         unsigned short dflag         : 1;
         unsigned short filler_1      : 15;
         unsigned short filler_2      : 8;
         unsigned short rflag         : 1;
         unsigned short filler_3      : 7;
         } keyparms[16];
      } ksamparam = { {0}, 0, {0}, {0,0,1,0,0,0,0}, 0, 3,
                      { {1, 5, 1,0,0,0,0,0},
```

```
                        {1,20, 6,1,0,0,0,0},
                        {1, 4,35,1,0,0,0,0} } };
   const record_t test_data[] =
      {
      "11111DOE JOHN            12300678985408212O1",
      "03452CUSTER HERB         32188000031608212O3",
      "28766WORKMAN DEBBIE      00066123415208506O1",
      "33678MORSE EUGENE        87600987631608507I5"
      } ;
   const int test_items = sizeof test_data / sizeof test_data[0];
   int i;
   /* First, purge file if it already exists */
   HPCICOMMAND("PURGE " FILENAME "\r", &cmderror, , 2);
   assert(!cmderror || cmderror==-383);
   /* Create new KSAM XL file, output access, 44-byte
      ASCII records, limit = 100, save disposition */
   HPFOPEN(&filenum, &status,
            2, "-" FILENAME "-",
           10, &ksamxl,
           11, &out,
           19, &recsize,
           35, &filesize,
           50, &save,
           53, &ascii,
           54, &ksamparam);
   assert(!status);
   /* Write test data to file */
   for (i=0; i<test_items; ++i)
      write_record(test_data[i]);
   printf("\n");
   /* Close file */
   FCLOSE(filenum, 0, 0);
   assert(ccode()==CCE);
}
static void delete_records(void)
{
   /* Delete records for several employees */
   const char delete_data[][5] = {"33678", "03452"};
   const int delete_items =
      sizeof delete_data / sizeof delete_data[0];
   int i;
   record_t buffer;
   for (i=0; i<delete_items; ++i)
      {
      printf("Deleting employee %.5s:  ", delete_data[i]);
      lock_file();
      FREADBYKEY(filenum, buffer, - sizeof buffer,
                 delete_data[i], 0);
      assert(ccode()==CCE);
      printf("%.20s\n", buffer+5);
      FREMOVE(filenum);
```

```
      assert(ccode()==CCE);
      unlock_file();
      }
   printf("\n");
}
static void dump_file(void)
{
   /* List the file several different ways */
   list_sequential_primary();
   list_sequential_secondary(6);
   list_sequential_secondary(35);
}
static void list_sequential(void)
{
   /* List the file, looping on FREAD until end-of-data */
   int save_ccode;
   record_t buffer;
   for (;;)
      {
      FREAD(filenum, buffer, - sizeof buffer);
      if ((save_ccode=ccode()) == CCG)
         break;
      assert(save_ccode==CCE);
      printf("     %.5s   %.20s   %.3s-%.2s-%.4s   "
             "%.4s   %.2s/%.2s/%.2s\n",
             buffer, buffer+5, buffer+25, buffer+28, buffer+30,
             buffer+34, buffer+40, buffer+42, buffer+38);
      }
   printf("\n");
}
static void list_sequential_primary(void)
{
   /* List file in sequence on primary key */
   printf("In sequence by primary key:\n");
   lock_file();
   /* Following call to FFINDN not necessary if this
      is the first access since the file was opened */
   FFINDN(filenum, -1, 0);
   assert(ccode()==CCE);
   list_sequential();
   unlock_file();
}
static void list_sequential_secondary(const int location)
{
   /* List file in sequence on specified secondary key */
   printf("In sequence by secondary key in location %d:\n",
          location);
   lock_file();
   FFINDN(filenum, -1, location);
   assert(ccode()==CCE);
   list_sequential();
```

```
   unlock_file();
}
static void lock_file(void)
{
   /* Lock the file unconditionally */
   FLOCK(filenum, 1);
   assert(ccode()==CCE);
}
static void open_file(void)
{
   /* Open file for shared update access with locking */
   int status;
   const int old=1, update=5, lock=1, shr=3;
   HPFOPEN(&filenum, &status,
            2, "-" FILENAME "-",
            3, &old,
           11, &update,
           12, &lock,
           13, &shr);
   assert(!status);
}
static void unlock_file(void)
{
   /* Unlock the file */
   FUNLOCK(filenum);
   assert(ccode()==CCE);
}
static void update_records(void)
{
   /* Update department code for several employees */
   const struct {char empno[5]; char new_dept[4];} update_data[] =
      {{"28766", "9901"}, {"11111", "9905"}};
   const int update_items =
      sizeof update_data / sizeof update_data[0];
   int i;
   record_t buffer;
   for (i=0; i<update_items; ++i)
      {
      printf("Updating employee %.5s to department %.4s:  ",
             update_data[i].empno, update_data[i].new_dept);
      lock_file();
      FREADBYKEY(filenum, buffer, - sizeof buffer,
                 update_data[i].empno, 0);
      assert(ccode()==CCE);
      printf("%.20s\n", buffer+5);
      memcpy(buffer+34, update_data[i].new_dept, 4);
      FUPDATE(filenum, buffer, - sizeof buffer);
      assert(ccode()==CCE);
      unlock_file();
      }
   printf("\n");
```

```
}
static void write_new_records(void)
{
   /* Add some entries to the file */
   const record_t test_data[] =
      {
      "77777NEWMAN GEORGE       7770066661520871012",
      "55555GOODMAN BRIAN       5553300008540880815",
      "66666MANLEY SHAUNA       0003526143360890930"
      } ;
   const int test_items = sizeof test_data / sizeof test_data[0];
   int i;
   for (i=0; i<test_items; ++i)
      {
      lock_file();
      write_record(test_data[i]);
      unlock_file();
      }
   printf("\n");
}
static void write_record(const char * const record)
{
   /* Write one record to the file */
   printf("Writing record for %.5s, %.20s\n", record, record+5);
   FWRITE(filenum, record, - sizeof(record_t), 0);
   assert(ccode()==CCE);
}
```

Index