HP 3000 MPE/iX Computer Systems

KSAM/3000 Reference Manual

HP Part No. 30000-90079
Printed in U.S.A. August 1986

Edition 2
E0886

PREFACE

This publication is the reference manual for KSAM/3000. KSAM stands for Keyed Sequential Access Method, a method of accessing files indexed by keys. KSAM/3000 operates on the HP 3000 Computer System.

The methods used to access a KSAM/3000 file differ depending on the particular language used. A COBOL user, an RPG user, a BASIC user, and an SPL user each has his own set of procedures with which to access a KSAM file; a FORTRAN user can choose to access a KSAM file with either COBOL or SPL procedures. All users can create, copy, purge, or perform other utility functions with the KSAMUTIL and FCOPY programs.

This manual is organized so that the more general functions available to all users are described in the first two sections followed by a section describing KSAM access from each of the four languages: COBOL, SPL, FORTRAN, and BASIC. Access to KSAM from a COBOL II program is through COBOL II's indexed I/O Module and is not documented in this manual. Access to KSAM files from an RPG program is not described in this manual, but is included as part of the RPG manual:

RPG/3000 Compiler Application & Reference Manual (32104-90001, Second Edition, 2/77)

In order to use this manual effectively, you should be familiar with the MPE Operating System and with FCOPY. Also, it is assumed that you are familiar with the language in which you are programming.

SECOND EDITION

The second edition of the KSAM manual provides the following new information:

- Full syntax for and description of how to use the new KSAMUTIL commands: KEYSEQ, KEYDUMP, and KEYINFO. (section II)
- Enhancements to the KSAMUTIL utility to allow abbreviated command names, offline listing of displays, and entry of MPE commands from KSAMUTIL. (section II)
- Discussion of record pointer positioning in all languages; with special emphasis on using the record pointers for shared access. (sections III, IV, VI)
- Description of how pointers are set internally. (appendix B)
- Discussion of recovery procedures in case of system failure. (appendix E)

In addition, there are minor corrections throughout the manual as well as documentation of minor enhancements.

This edition covers the version of KSAM number A.02.04 release on the 1918 IT.

Contents

3. USING KSAM FILES IN COBOL PROGRAMS

Figures

Tables

1

INTRODUCING KSAM/3000

OVERVIEW

The Keyed Sequential Access Method (KSAM) is a method of organizing records in a file according to the content of key fields within each record. As implemented for the HP 3000 computer system, KSAM/3000 is similar to and competitive with other indexed sequential access methods.

Every record in a KSAM file contains a primary key field whose contents determine the primary logical sequence of records in the file. Other key fields can also be defined so that the file can be sequenced in alternate orders. The order in which records are physically written to the file, the chronological order, can be the same as the primary key sequence or it can be unrelated to any logical sequence.

KSAM/3000 files can be accessed by programs written in any of these languages:

RPG/3000
COBOL II/3000
COBOL/3000
SPL/3000
FORTRAN/3000
BASIC/3000

KSAM/3000 files can be copied, listed, and otherwise manipulated with the utility programs:

FCOPY/3000
KSAMUTIL

FILE STRUCTURE

A KSAM/3000 file is organized into two distinct MPE files, a data file and a key file. The key file contains only key entries, the data file only data. Each record in the data file contains at least one item that is designated as a key. The value of each key is duplicated in the key file where all keys are ordered in ascending sequence. This organization allows records in the data file to be stored in any order since the key file maintains the logical order of records according to key value.

Although it is not necessary to understand KSAM file structure in order to use a KSAM file, you may want to refer to appendix B for a detailed discussion of the relation between data and key files and the structure of the key file.

Note Each open KSAM file uses three file numbers; two for the data and keys and one for internal maintenance. The maximum number of KSAM files that can be opened for a process depends upon the stack of that process.

FILE ACCESS

Although separate in fact, the two files that comprise a KSAM file are treated as one file by the procedures that reference the file. The data file is the only file directly referenced by a user; the key file is updated by the system to reflect any changes to the data file and is not directly accessed by the user. *Thus, from the user's point of view, accessing a KSAM file is very similar to accessing any other MPE file.*

KSAM/3000 provides the following ways to store and retrieve data:

- You can write records in logical sequence determined by primary key value or you can write records without regard to key sequence.
- You can read records in logical sequence determined by either the primary or an alternate key value.
- You can read a record selected at random by the value of its primary or alternate keys.
- You can read records in the order they were written, that is, in chronological sequence, unless the program is written in COBOL or BASIC.
- You can read a record selected by the value of its chronological record number, unless the program is written in COBOL or BASIC.
- You can update all the contents of an existing record including the contents of the primary key field.
- You can position to a record in the file according to its key value, its chronological record number, or its record number in key sequence.

Note KSAM files are sequenced in *ascending* order only, not in descending order. Character keys are ordered by the ASCII collating sequence where numbers precede letters, not in the EBCDIC sequence where letters precede numbers. Numeric keys are ordered in algebraic order.

KSAM/3000 FEATURES

KSAM/3000 provides a number of features beyond the standard indexed sequential access method. These include:

- Multiple Keys
- Duplicate Key Values
- Retrieval by Generic Key
- Retrieval by Approximate Match
- Fixed or Variable Length Data Records

MULTIPLE KEYS

Each data record can contain from one to sixteen keys. Of these keys, one is required, called the primary key; any others are alternate keys. For example, in an employee record, the primary key could be the employee's social security number; alternate keys might be the employee's name, phone number, or zip code. The values in these key fields determine the orders in which data records are sequenced.

PRIMARY KEY. One field in each data record is defined to contain the primary key. The value in this field determines the primary sequence of records in the data file. Records are sequenced according to this primary key unless sequencing by an alternate key or in chronological order is specifically requested.

ALTERNATE KEYS. Other fields within each data record can be designated as alernate keys to be used for alternate sequencing of records. Up to 15 alternate keys can be designated for each record, however, each additional alernate key adds to the overhead and can affect performance when accessing and maintaining a file. The file can be sequenced in a different order for each alternate key defined for the file.

Note that alternate keys bear no hierarchical relation to each other or to the primary key. Each key is ordered in sequence by its value and type with no relation to other keys. In KSAM, sequence always means ascending sequence according to the ASCII collating sequence, (refer to appendix C.)

DUPLICATE KEYS

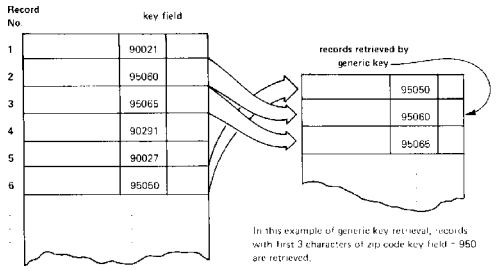
Sometimes it is essential that key values be unique (for example, a social security number), and at other times duplicate key values should be allowed (for example, a zip code). To provide for both cases, KSAM allows you to declare that any key may have a duplicate value while disallowing duplicate key values as the default condition. Allowing or disallowing duplicate key values applies to both primary and alternate keys. Duplicates can be allowed for one or more keys while being disallowed for other keys.

Note Duplicate keys can greatly increase the time required to load or access a record with a duplicated key value. This is particularly true when there are a large number of duplicated key values in a large file. As a result, duplicate keys should only be used when other methods are not practical. For example, you should not make a key of an item that can only have two values, as "MALE" or "FEMALE."

GENERIC KEYS

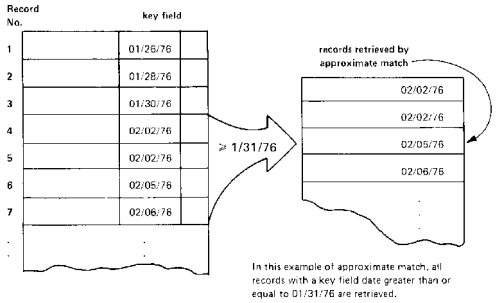
During retrieval by key value you can choose to use part of a key rather than the entire key. Called generic keys, such partial keys allow you to retrieve a set of records whose key values differ in their entirety but share a common value at the beginning. Generic keys must begin at the first character of the defined key held and be shorter, not longer, than the defined key length; also, the key type must be BYTE, INTEGER, or DOUBLE. Suppose a key field containing a zip code is defined as five characters long. By specifying only the first three characters for retrieval it is possible to read all records whose zip code begins with a particular group of numbers.

Note Generic keys cannot be used when accessing KSAM files through RPG.



APPROXIMATE MATCH

When retrieving by key value, you can specify that the key you are looking for have a value that exactly matches a specified value, or you can specify that it bear a certain relation to a specified value. The choices are: equal to, equal to or greater than, or greater than. The last two relations let you search for an approximate match. For example, you can retrieve all records with a date greater than or equal to a given date:



DATA RECORD FORMAT

Every key entry in the key file contains, in addition to the key value, a pointer to the corresponding data record in the data file. The data records can be either fixed length or variable length. If they are fixed, the data record pointer specifies a record number relative to the beginning of the file. If the records are variable length, then the pointer indicates the start of the data record as a word offset from the beginning of the file.

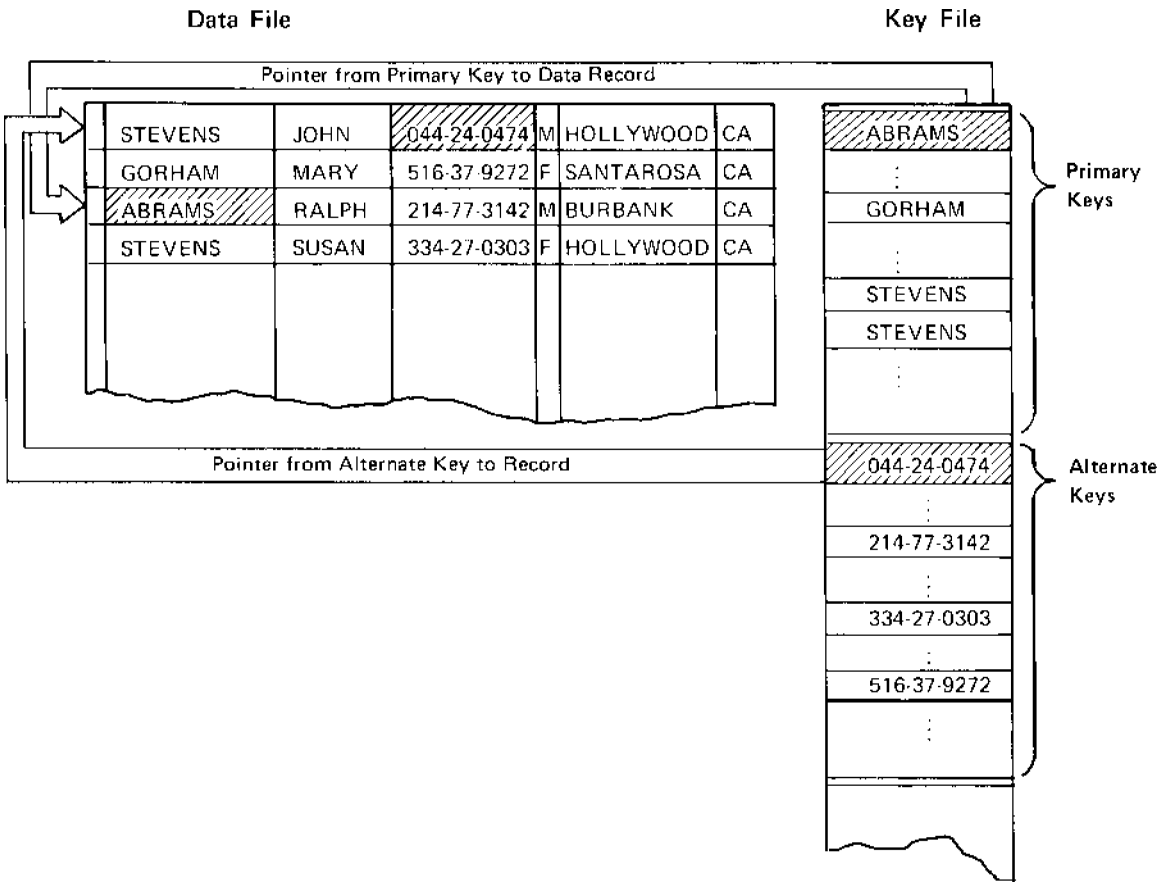


Figure 1-1. A Simplified View of the KSAM File Structure

HOW TO USE KSAM FILES

Although a KSAM file consists physically of two separate files, a data and a key file, it is treated as one file for most purposes. For example, reading from a KSAM file in primary key sequence is equivalent to reading sequentially from a non-KSAM file. Similarly, creating the data file portion of a KSAM file is equivalent to creating a non-KSAM file.

CREATING A KSAM FILE

A KSAM file can be created in two ways: interactively with the > BUILD command of the utility program KSAMUTIL, or programmatically with a call to the MPE file system intrinsic FOPEN. (A COBOL or BASIC programmer can create a KSAM file only through the > BUILD command, not FOPEN.) Whether > BUILD or FOPEN is used, file creation consists of creating a data file in very much the same way you would create any HP 3000 file. The name assigned to the data file is the name by which the KSAM file is known. Then, as part of the file creation procedure, a key file is created and each of its keys defined by type, location in the data record, and size. If duplicate key values are to be allowed, this is specified as part of the key definition.

WRITING RECORDS TO A KSAM FILE

You can write records to a KSAM file in either of two ways. In one, records are written in any order regardless of primary key values. In the other, records are written in order according to the value of the primary key in each record. In the first case, the chronological sequence in which records are written differs from the logical record sequence determined by primary key. In the second, the chronological and logical record sequence is the same. When you specify that records are to be written in primary key sequence, KSAM checks to make sure that this sequence is followed and issues an error message if not.

You can specify that the file be cleared of any existing records before writing new records to the file, or you can write records following any previously written records. The choice is made when you open the file.

In any case, when records are written to the data file, the key file structure is modified automatically in order to place all keys in the new record into their proper sequence.

Records cannot be written directly to a KSAM file according to a relative record number.

RETRIEVING RECORDS FROM A KSAM FILE

Records can be retrieved in a variety of ways:

- Sequentially in the order determined by key value; either the primary or an alternate key can be selected to determine the order.
- At random according to the value of a specified key; either the primary or an alternate key can be selected for the matching process.
- Chronologically in the order the data records were written.*
- At random by chronological record number.*

*The starred access methods are not available to a COBOL or BASIC programmer.

Whenever duplicate keys are used and retrieval is by key value, the first key encountered determines the record read. When generic keys are used, the smallest key value is selected first. Again, if there are duplicates in generic key values, the first key encountered is selected.

UPDATING RECORDS IN A KSAM FILE

You can change the contents of an existing record by program calls that read the record into storage where you update it and then write it back to the file. The updated record overwrites the existing record in its current location if the new record and the old record are the same length. Otherwise, the new record is written to the end of the file and the old record is marked for deletion.

POSITIONING IN A KSAM FILE

Record pointers can be positioned:

- To a record determined by key value using either the primary or an alternate key.
- To a record determined by its record number relative to the first record in key sequence, where the key is either primary or an alternate.**
- To a record determined by its record number relative to the first record written to the file (chronological sequence).**

**Not available in COBOL or BASIC program.

DELETING RECORDS FROM A KSAM FILE

Records are not physically deleted from the data file. In order to delete a record, you call a procedure that marks the record for deletion by writing a delete code (octal 177777) in the first word of the record. Any subsequent access to the file skips such records, treating them as if they were not there. In addition, the key file is automatically reorganized so that keys in the deleted record are no longer in the path that defines the key sequence. The space in the key file created by the deleted entries is reused. However, to maintain the file's chronological order, the space occupied by deleted entries in the data file is not reused.

Because data records are not physically deleted, you can reconstruct deleted records by copying the data file using FCOPY with the NOKSAM and SUBSET options. In this way, you can restore files in which you deleted, records by mistake.

To prevent records from being deleted by mistake, do not place binary or numeric data that could result in octal 177777 (the deletion code) in the first word of a data record. The best practice is to always keep the first word of each data record blank. Doing so helps you identify deleted records and prevents KSAM from writing delete codes over recoverable data.

REORGANIZING A KSAM FILE

If many records have been deleted, thereby using a great deal of physical space in the file, you can compact the file by using FCOPY/3000 to copy only the active records, those not tagged for deletion, to a new KSAM file. You can also use FCOPY to delete, add, or change alternate keys by copying the file to a new KSANI file with a different key definition. When the key definition is different, you must first create the new file with the >BUILD command of KSAMUTIL.

SHARED ACCESS TO KSAM FILES

Several programs can access the same KSAM file simultaneously. Shared access is assumed when the file is only being read, exclusive access is assumed when the file is being written to or updated. Thus, you can choose to make all your access shared or all exclusive. Note that shared access uses more memory than exclusive access since each open KSAM file requires a separate extra data segment.

When access to the file is shared, it is each user's responsibility to dynamically lock the file before changing it in any way. The file must be locked before any records in the file are written, updated, or deleted, and then unlocked immediately after such action. By requiring this action, the system makes sure that the most recent values are brought into each user's buffer at each access. Any call to read or position to a record for sake of subsequent access should be within the locked portion of code that includes the actual update call.

(Refer to appendix E for a full discussion of shared access.)

RECOVERY AND ANALYSIS OF KSAM FILES

The utility program KSAMUTIL provides several commands that can be used to analyze KSAM files. These commands allow you to check any key sequence to obtain a formatted dump of the key file, and in the event of a system failure, to check key file structural damage, determine whether key values are missing, and recover key values and data records by resetting end-of-file pointers. The command, KEYINFO, that performs these recovery functions must be run in case of a system failure while a KSAM file is open. (A full discussion of these commands is found in section II; also refer to appendix E for a discussion of KSAM file recovery iii the event of system failure.)

USING FILE EQUATIONS WITH KSAM FILES

KSAM opens the key and data file allowing file equations for both. KSAM accesses the files in a very specific way. Since file equations will override any aoptions that KSAM uses, it is possible to specify access parameters on the file equation which will cause unpredictable results. The following should not be used on a file equation which references a KSAM file:

```
BUF=numbers of buffers
NOMR
WAIT
```

Refer to "Dynamic Locking" and "Exclusive Access" in Table 4-7, and the section on using FCOPY to add data to an existing file for further information.

HOW TO USE THIS MANUAL

There are some differences in the way in which KSAM files can be accessed depending on the language in which you are programming. You should read the paragraphs below appropriate to your programming language and then turn to the last paragraph of this section, For All Programmers.

RPG PROGRAMMER

This manual does not describe the code required to access a KSAM file using RPG. For this information, you must refer to:

RPG/3000 Compiler Applications & Reference Manual

COBOL II PROGRAMMER

If you are programming in COBOL II, you can access KSAM files through COBOL II's Indexed I/O Module. To do so you declare a file's organization to be INDEXED. Refer to the *COBOL II Reference Manual* (32233-90001) for documentation of this procedure.

You can optionally access KSAM files using MPE Intrinsics (Section IV) and COBOL procedures (Section III).It is recommended that you use COBOL procedures only when compatibility with COBOL is required.

COBOL PROGRAMMER

If you are programming in COBOL, you should read section II in order to learn how to:

- Create, purge, rename, clear the contents, display the status of, or save a KSAM file. These functions are provided by the KSAMUTIL program.
- Copy a KSAM file to another KSAM file in any key order.
- Display the contents of a KSAM file in any key order on the standard list device. These functions are provided by the FCOPY program.

You should read section III in order to learn how to:

- Open and close the KSAM file.
- Open the file for shared access and dynamic locking.
- Write the records to the file in sequential key order or in random order.
- Read records from the file in sequential order by key value or at random by key value.
- Change the key in preparation for a sequential read.
- Rewrite or delete an existing record.
- Dynamically lock or unlock the file.

Note the following limitations for COBOL:

- You cannot programmatically create a KSAM file. You must use the > BUILD command of the KSAMUTIL utility program in order to create the file.
- You cannot read a KSAM file in chronological sequence. You can, however, use FCOPY to copy the file to a non-KSAM file and then read it in chronological sequence.

- For ANSII standard COBOL, only alternate keys, not primary keys, can be duplicated.

SPL PROGRAMMER

If you are programming in SPL, you should read section II in order to learn how to:

- Create, purge, rename, clear the contents, display the status of, or save a KSAM file. These functions are provided by KSAMUTIL.
- Copy a KSAM file to another KSAM file in any key order.
- Display the contents of a KSAM file in any key order on the standard list device. These functions are provided by FCOPY.

You may skip sections III, V, and VI, which apply to programming in COBOL, FORTRAN, and BASIC respectively. You should read section IV to learn how to:

- Create, open, and close a KSAM file.
- Write records to the file in sequential primary key order or in random order.
- Read records from the file in primary or altenate key order or in chronological order.
- Read records at random by key value.
- Read records directly according to a record number relative to the first chronological record.
- Position record pointer forward or backward a specified number of records in any specified key sequence.
- Position to a record defined by key value.
- Position to a relative record number in key sequence or in chronological sequence.
- Update or delete an existing record.
- Request access and status information on the KSAM file.
- Verify that input/output is completed, and verify that critical output is complete.
- Dynamically lock or unlock the file.
- Write or read user labels.

In general, SPL programmers can use all the file system intrinsics provided for HP 3000 standard files with the following exceptions:

- A KSAM file cannot be renamed with the FRENAME intrinsic.
- A KSAM file cannot be positioned to a relative record number with FREADSEEK. (Similar functions are performed by the KSAM intrinsics FFINDBYKEY and FFINDN).
- A record cannot be written to a KSAM file according to relative record number with FWRITEDIR.
- The relation between two files (interactive or duplicative) cannot be determined with FRELATE.

FORTRAN PROGRAMMER

If you are programming in FORTRAN, you should read section II in order to learn how to:

- Create, purge, rename, clear the contents, display the status of, or save a KSAM file using KSAMUTIL.
- Copy a KSAM file to another KSAM file in any key order with FCOPY.
- Display the contents of a KSAM file in any key order on the standard list device using FCOPY.

As a FORTRAN programmer can call either the COBOL procedures described in section III (and summarized above) or the intrinsics described in section IV (also summarized above). You should, therefore, read both these sections. Depending on your program requirements, you can then choose to use either the COBOL procedures or the file system intrinsics. Since these methods differ significantly in how the file is created and accessed, you should not attempt to combine calls to COBOL procedures with calls to the file system intrinsics. In general, the intrinsics provide more capabilities than the COBOL procedures.

You should also read section V, which illustrates, by means of annotated examples, how to access a KSAM file through FORTRAN calls to the file system intrinsics. The examples illustrate:

- Programmatically creating a KSAM file.
- Writing records to a new KSAM file.
- Reading the records in sequential order by primary key value and then by alternate key value.
- Reading the records in chronological order.

BASIC PROGRAMMER

As a BASIC programmer you should read section II in order to learn how to:

- Create, purge, rename, clear the contents, display the status of, or save a KSAM file using KSAMUTIL.
- Copy a KSAM file to another KSAM file in any key order with FCOPY.
- Display the contents of a KSAM file in any key order on the standard list device using FCOPY.

Since a BASIC programmer, like the COBOL programmer, cannot create a KSAM file programmatically, it is especially important to note how files are created with the BUILD command of program KSAMUTIL. Note also that BASIC programs cannot read a KSAM file in chronological sequence. You can, however, use FCOPY to copy the data file to a non-KSAM file and then read it in chronological sequence.

You can skip sections III, IV, and V, which apply to COBOL, SPL, and FORTRAN programming respectively, and read section VI, which describes the BASIC procedures to access KSAM files. These procedures enable you to:

- Open and close a KSAM file.
- Write records to a KSAM file in primary key or in random order.
- Read records from the file in sequential order by key value, or at random by key value.

- Change the key in preparation for a sequential read.
- Rewrite or delete an existing record.
- Dynamically lock and then unlock the file during shared access.

ALL PROGRAMMERS

Programmers using any of the languages that access KSAM files will probably need to refer to appendix A. This appendix contains an explanation of the error messages, condition codes, and status returns that can result from file access.

Appendix B describes the internal structure of KSAM files. It illustrates how key entries are stored in a special B-Tree structure, and how KSAM file size is determined. It also explains how files are accessed through the extra data segments allocated to each open file. This appendix provides information for the sophisticated programmer who wants to know how KSAM files operate in order to improve performance. For the average user, the information in appendix B is not needed in order to create and use KSAM files.

Appendix C provides the ASCII collating sequence used by KSAM /3000 to determine character key sequence; (numeric key sequence is in algebraic order). Note that the KSAM key sequence is in ascending order only, the order in which the ASCII characters are shown in appendix C.

Appendix D provides instructions that will help you convert your files to KSAM /3000 files. It tells you how to convert any serially accessible file to a KSAM file. If you are already using INDEX files, it describes use of the conversion program RTOKSAM for converting from INDEX to KSAM. Note that INDEX files were previously called RSAM files.

Appendix E describes the recovery procedures to be used if the system fails when KSAM files are open. It explains what happens when a file is closed normally as opposed to what happens when a system failure prevents normal closing, and then tells the user exactly what to do when a system failure affects open KSAM files.

2

USING KSAM UTILITIES

USING KSAM UTILITIES

A pair of utility programs and a set of commands allow you to create and manipulate KSAM files.

OVERVIEW

The program KSAMUTIL provides MPE capabilities that allow you to manipulate KSAM files. With KSAMUTIL commands, you can create a KSAM file, rename both the data and key files, save a temporary file as a permanent file, clear all data from a file, purge a file, and verify the contents and access history of an existing file.

The HP 3000 file copier, FCOPY, is adapted to copy KSAM files. FCOPY allows you to copy from a KSAM file to another file (KSAM or non-KSAM), in primary or alternate key sequence; to copy an entire file or a subset of a file, and to copy either the data or key file.

The MPE commands :STORE and :RESTORE can be used with KSAM files to transfer the files from disc to magnetic tape and vice versa.

The utility functions that can be performed on KSAM files are summarized in Table 2-1.

Both KSAMUTIL and FCOPY are programs resident in the system library that can be executed with the MPE :RUN command. When run in a session, each program responds by issuing a greater-than (>) prompt. You may then enter commands to control further operation of the program. Both programs may be operated in batch mode as well as in a session. In batch mode, the greater-than prompt is not required. :STORE and :RESTORE are commands directed to the MPE command interpreter and can be included in either a job or a session.

Table 2-1. Summary of KSAM Utilities

UTILITY	OPTION/COMMAND	FUNCTION
KSAMUTIL	>BUILD or >B	Create KSAM file consisting of a data file and key file.
	>ERASE	Clear contents of KSAM data file and reset key file pointers.
	>PURGE	Remove KSAM file from system.
	>RENAME or >R	Change name of KSAM key or data file to a new name.
	>SAVE or >S	Save session/job temporary KSAM file as a permanent file.
	>VERIFY or >V	Display information on current status of data and keys in KSAM file.
	>HELP or >H	Request description of KSAMUTIL commands.
	>EXIT or >E	Exit from KSAMUTIL program.
	>KEYSEQ or >KS	Check the sequence of any key (primary or alternate) in key file.
	>KEYDUMP or >KD	Display a formatted, structural key file dump.
	>KEYINFO or >K1	Display information on current status of key file; in case of system failure, attempt recovery.
FCOPY	;KEY=*keylocation*	Copy KSAM file in key sequence by a key specified by its beginning location in record.
	;NOKSAM	Copy contents of key or data file in consecutive (physical) order. If both these parameters are omitted, the data file is copied in sequence by primary key; the key file is established with all links maintained. Other FCOPY options apply to KSAM files with minor exceptions (refer to Table 2-4).
MPE	:STORE *datafile, keyfile*	Store KSAM data and key files from disc to magnetic tape.
	:RESTORE *datafile, keyfile*	Restore KSAM data and key files from magnetic tape.

KSAMUTIL UTILITY

KSAMUTIL provides a number of capabilities, among which is the essential capability to create KSAM files. For a COBOL, BASIC, or RPG programmer, KSAM files can be created only through the BUILD command of the program KSAMUTIL. Although SPL and FORTRAN programmers can create KSAM files with the FOPEN intrinsic (described in section IV), the BUILD command may still provide these users with the simplest method for creating a KSAM file.

RUNNING KSAMUTIL

To pass control to KSAMUTlL, use the MPE command:

```
:RUN KSAMUTIL.PUB.SYS
```

In a session, KSAMUTIL prompts with the greater-than sign (>) in column 1 to which you respond with the command you want to execute. In a job, you enter the command in column 1 of the record following the RUN command. No prompt character precedes the KSAMUTIL commands in batch mode.

Refer to Table 2-1 for a list of the KSAMUTIL commands and their functions.

COMMAND ABBREVIATIONS.

All KSAMUTIL commands, except ERASE and PURGE, can be abbreviated. Most abbreviations allow the first letter of the command name. For example, >BUILD can be specified as >B, and >EXIT can be specified as >E. The three command names beginning with K (>KEYDUMP, >KEYSEQ, and >KEYINFO) are abbreviated to two letters to distinguish one from the other. As shown in Table 2-1, these abbreviations are, respectively, >KD, >KS, and >KI.

RUNNING MPE COMMANDS FROM KSAMUTIL.

Once you are running KSAMUTIL and you want to use an MPE command, you need not exit from KSAMUTIL and return to MPE; simply type the colon prompt (:) following the KSAMUTIL prompt (>) and then enter the MPE command. For instance, if you want to list the files in your account and group from KSAMUTIL, enter the LISTF command as shown:

```
>:LISTF
```

OPTION TO LIST DISPLAYS ON LINE PRINTER.

Four KSAMUTIL commands display file information; these are VERIFY, KEYDUMP, KEYSEQ, and KEYINFO. Each of these has an option that allows you to list the information on a line printer rather than display it on your terminal. If you include the keyword OFFLINE as an option in any of these commands, the requested information is sent to the line printer. If you want the list sent to a particular line printer, you can use a :FILE command naming the KSAM list file "KSAMLIST" as the formal designator. For example, suppose you are running KSAMUTIL and want to list the current information on a KSAM file and you want this information listed on a particular line printer:

```
>:FILE KSAMLIST; DEV=SLOWLP <----                                  select particular line printer
>VERIFY MYFILE; OFFLINE<---------                                  specify output to go to an offline device
WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)? 4   request all information———-
```

The resulting output is sent to the line printer identified as SLOWLP.

OPTIONAL PARAMETERS.

Wherever a command parameter is shown with brackets, [], that parameter can be omitted. For certain commands, SAVE, VERIFY, KEYDUMP, KEYSEQ, and KEYINFO, the *filereference* parameter is optional if no other parameters are specified. When this parameter is omitted, it assumes a prior command has specified a *filereference* and it uses the last *filereference* to identify the selected file. For example, assume you use the VERIFY command twice in a row, once to list the requested output on the line printer, and then to display it at your terminal. To do this, you can use the following command sequence:

```
:RUN KSAMUTIL
>VERIFY MYFILE; OFFLINE
WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?4

(output is sent to the line printer)

>V<---------------------------- previous file reference to MYFILE is assumed
WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)? 4

  (output appears at your terminal)
```

Note that you cannot issue these commands in reverse order because the *filereference* parameter can be omitted only if there are no other parameters. Thus, it is *not* legal to use command >VERIFY MYFILE followed by >VERIFY; OFFLINE.

EXITING FROM KSAMUTIL

When you have finished using KSAMUTIL in a session, you can return to the MPE operating system with the command:

```
>EXIT or >E
```

In a batch job, the EXIT command is specified in column 1 of the record that terminates the program; the greater-than sign is not included. The keyword EXIT can be abbreviated as E.

HELP

Requests help using KSAMUTIL

```
>{ HELP }
 { H    }
```

This command returns a summary description of each of the KSAMUTIL commands when entered at a terminal. The keyword "HELP" can be abbreviated to "H" .

REQUESTING HELP

The HELP command lists all the valid KSAMUTIL commands and then asks if you need information on a particular command. When you enter the name of a KSAMUTIL command, HELP displays the command syntax. HELP is terminated by entering N in response to the prompt MORE (Y/N)?.

The following example illustrates the HELP command.

```
>H

VALID COMMANDS ARE:
BUILD[B].......TO CREATE A FILE
ERASE..........TO RESET A FILE TO INITIAL CONDITIONS
EXIT[E]........TO LEAVE THIS ROUTINE
PURGE..........TO DELETE A FILE
RENAME[R]......TO RENAME A FILE
SAVE[S]........TO RETAIN A TEMPORARY FILE
VERIFY[V]......TO DESCRIBE FILE CHARACTERISTICS
KEYINFO[KI]....TO OBTAIN KEY FILE INFORMATIONS
KEYDUMB[KD]....TO OBTAIN FORMATTED KEY FILE STRUCTURAL DUMP
KEYSEQ[KS].....TO CHECK THE SEQUENCE OF KEY FILE

MORE (Y/N)?Y

ENTER COMMAND NAME: B
BUILD  <DATAFILEREF>
[;DEV= <DEVICE>]
[;DISC=[ <NUMREC>][, <NUMEXTENTS>][, <INITAL-OC>]]]
(REC=[ <RECSIZE>][,[ <BLOCKFACTOP>][,[F \V][,BINARY \,ASCII]]]]
[;TEMP]
[;CODE= <FILECODE>]
;KEY= <TYPE>, <POSITION>[,[ <LENGTH>][,[ <BLOCKING>][,DUPLICATE]]]
                                                    [,RDUP]
[;KEY= <TYPE>, <POSITION>[,[ <LENGTH>][,[ <BLOCKING>][,DUPLICATE]]] ...]
                                                    [,RDUP]]]...]
[;LABELS= <NUMBERLABELS>]
[;FIRSTREC=0\-1]
[;KEYDEV= <DEVICE>]
;KEYFILE =FILEREFERENCE2
[;KEYENTPIES= <NUMBER>]
 <TYPE>::=B\D\I\R\L\N\P\*

MORE (Y/N)?Y

ENTER COMMAND NAME: E

EXIT

MORE (Y/N)?Y

ENTER COMMAND NAME: ERASE
```

```
ERASE  <FTLEREFERENCE>

MORE (Y/N)?Y

ENTER COMMAND NAME: PURGE

PURGE  <FILEREFERENCE>[,TEMP]

MORE (Y/N)?Y

ENTER COMMAND NAME: R

RENAME  <OLDFILEREF>, <NEWFILEREF>[,TEMP]

MORE (Y/N)?Y

ENTER COMMAND NAME: S

SAVE [ <TEMPFILEREF>]

MORE (Y/N)?Y

ENTER COMMAND NAME: V

VERIFY [ <FILFREFERENCE>]
[;OFFLINE]
[;NOCHECK]
MORE (Y/N)?Y

ENTER COMMAND NAME: KD

KEYDUMP [ <FILEREFERENCE>]
[;SEQ= <KEYSEQUENCE>]
[;SUBSET=[ [ minus;] <POSITION>][, <NUMBER>]]
         I                  I
         ["CHAR-STRING"]
[;FILE = <FILEREFERENCE1>]
I                      I
[;OFFLINE              ]
[;SORT]

MORE (Y/N)?Y

ENTER COMMAND NAME: KI

KEYINFO [ <FILEREFERENCE>]
[;OFFLINE]
[;RECOVER]
MORE (Y/N)?Y


ENTER COMMAND NAME: KS

KEYSEQ [ <FILEREFERENCE>]
[;SEQ= <KEYSEQENCE>]
(;OFFLINE]
[;NOLIST]
MORE (Y/N)?N <---------------- terminate HELP display

>E

END OF PROGRAM
```

BUILD

Creates a KSAM file.

```
> B
BUILD filereference 1
[;REC=[recsize] [,[blockfactor] [,F  ,BINARY
                                    V  ,ASCII] ] ]


[;TEMP]
[;DEV=device]
[;CODE=filecode]
[;DISC=[numrec] [,[numextents] [,initialloc] ] ]

;KEYENTITY =filereference 2
                                                ,DUPLICATE
                                                ,DUP
;KEY=keytype, keylocation, keysize[,[keyblocking] ,RDUPLICATE
                                                ,RDUP

                                                  ,DUPLICATE
;KEY=keytype, keylocation, keysize[,[keyblocking] ,DUP
                                                  ,RDUPLICATE
                                                  ,RDUP

[;KEYENTRIES=numentries]
[;LABELS=numlabels]
[;KEYDEV=device]
[;FIRSTREC=recnum]
[;LANG=parameter]
[;NODUPLOCKS]
```

The BUILD command of the KSAMUTIL utility program is used to create a KSAM file and allocate the file to a mass storage device. Although this command is similar to the MPE :BUILD command, it has been modified for KSAM files. You can specify the BUILD command with the abbreviation, B.

Note You cannot create a KSAM file with the MPE :BUILD command.

If you are programming in COBOL, BASIC, or RPG, you must use the KSAMUTIL BUILD command to create a KSAM file; in SPL or FORTRAN, you can create a KSAM file either with the BUILD command or with the FOPEN intrinsic (described in section IV).

PARAMETERS

filereference1 Actual file designator. This is the name that identifies the KSAM file (both data and key files) and also identifies the data file when specified independently of the key file. It has the form:

filename [/lockword] [.groupname [.accountname]]

All four sub-parameters are names that contain from 1 to 8 alphanumeric characters, beginning with a letter.

Note If specified, account name must be that of your log-on account; *you cannot create a file in another account.*

If file has no lockword and belongs to your log-on group, only *filename* is necessary. You cannot backreference files.

(Required parameter.)

REC=*recsize* Size of logical records in file. If a positive number, this represents words; characters are represented by a negative number, If the records are variable length, *recsize* indicates the maximum length allowed for a logical record.

Block size is determined by multiplying the specified *recsize* by *blockfactor*. For binary files or ASCII files with fixed-length records, an odd character count is rounded up to the next highest even number to insure that the record starts on a word boundary. The rounded number should be used in calculating block size since a block always starts on a word boundary.

(Optional parameter.)

Default: The configured record size of the particular device is used when recsize is omitted; for disc files, the value used is 256 characters or 128 words.

blockfactor An integer equal to the number of logical data records in each block. This integer should result in a data block size smaller than 4096 (4K) words. The *blockfactor* is used to calculate the buffer size established for transfer of data to and from the file.

For fixed-length records, *blockfactor* is the actual number of records in a block. For variable-length records, *blockfactor* is a multiplier used with *recsize* to calculate block size:

block size = ((*recsize*+1) * *blockfactor*)+1

The calculation is performed in words, not characters.

(Optional parameter.)

Default: calculated by dividing the specified recsize into the configured block size; the result is rounded down to an integer never less than 1.

F Data file contains fixed-length records.

(Optional parameter.)

V Data file contains variable-length records. Since KSAM performs its own blocking and deblocking, a KSAM data file specified as variablelength is

treated by MPE as a file with fixed-length records, each record the size of a KSAM block (refer to *blockfactor* above for calculation of block size). Although the MPE LISTF command shows the data file as fixed-length, the KSAMUTIL VERIFY command, option 3, shows DATA FIXED as FALSE when the file is a variable-length KSAM file.

(Optional parameter.)

Default: If both F and V are omitted, records are fixed-length.

BINARY — Data file contains binary-coded records.

(Optional parameter.)

ASCII — Data file contains ASCII-coded records.

(Optional parameters.)

Default: If both BINARY and ASCH are omitted, records are binary.

TEMP — File is created as a session/job temporary file; when the session or job terminates, the file is deleted from the session/job temporary file directory.

(Optional parameter.)

Default: If TEMP is omitted, file is declared permanent and is saved in the system file domain.

DEV=*device* — *device* designates the device on which the data file resides. (The key file device is specified in the KEYDEV parameter.) *device* can be specified as a device class name of up to 8 alphanumeric characters beginning with a letter and terminated by any non-alphanumeric character such as a blank, or as a logical device number consisting of a three-character numeric string, or it can be a remote device identifier consisting of the device class name or logical device number followed by a pound sign (#) and a remote device class name or logical device number.

Device class names and logical device numbers are assigned to devices during system configuration. (See *System Manager/System Supervisor Reference Manual*).

For KSAM files, the device must be a random access device such as the disc. If the file is a newly-created disc file specified as a device class name, then all extents to the file must be members of the same class. Similarly, if the device is identified by a logical device number then all extents must have the same logical device number.

(Optional parameter.)

Default: If omitted, the device class name DISC is used.

CODE=*filecode* — Code indicates that the data file is specially formatted. The code is recorded in the file label and is available to processes through the FGETINFO intrinsic. It must be specified as a positive integer in the range 0 through 1023.

(Optional parameter.)

Default: If CODE is omitted, the file code is 0.

Note The CODE parameter applies only to data files; the key file code value is always 1080.

DISC=*numrec* Total maximum file capacity, in terms of logical records (for files containing fixed-length records). For files containing variable-length records, this is the maximum file capacity if all the records are maximum length. Maximum file capacity allowed is 2,097,120 sectors.

(Optional parameter.)

Default: If omitted, 1024 records is the default.

numextents Number of extents (continguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent (in terms of records) is determined by the *numrec* parameter value divided by the *numextents* parameter value. Extents can allocated on any disc in the device class specified in the *device* parameter. If you want to ensure that all extents for a file reside on the same disc, use the logical device number of that disc or a device class name relating to a single disc device, in the *device* parameter. If specified, *numextents* must be integer value from 1 to 32.

(Optional parameter.)

Default: 8.

initalloc Number of extents to be allocated to the file *at the time it is opened*. Must be an integer from 1 to 32. If attempt to allocate requested space fails, an error message appears.

(Optional parameter.)

Default: 1.

KEYFILE =*filereference2* Actual file designator. This is the name that identifies the KSAM key file. It has the format: *filename*, which is 1-8 alphanumeric characters beginning with a letter. Unlike *filereference1* (the data filename) *filereference2* may not be qualified by account or group names, nor may it contain a lockword. The key file contains all the key entries and key control information, whereas the data file contains the actual data. A KSAM file is always referenced by the data file name, *filereference1*, not the key file name, *filereference2*.

(Required parameter.)

KEY= One KEY specification must be included for each key in the KSAM file. The first occurrence of the KEY specification describes the primary key; each subsequent KEY specification describes an alternate key. There may be up to 15 alternate key descriptions in addition to the primary key description.

(Required parameter.)

keytype *keytype* is specified as BYTE,INTEGER, DOUBLE, REAL, LONG, NUMERIC, PACKED, or *PACKED. The whole word or only the first letter need be specified (for example, B is equivalent to BYTE). If more than the first letter is used, the word must be spelled correctly. (Refer to Table 2-2 for a full description of each key type.) (Required parameter.)

keylocation Location of the first character of the key within the data record counting from the first character in the record. The first character in the data record is always numbered 1. Only one key can start at the same location. (Required parameter.)

keysize Length of the key in characters. The length depends on *keytype* as follows:

BYTE	1 to 255 characters
INTEGER	1 to 255 characters (default = 2)
DOUBLE	1 to 255 characters (default = 4)
REAL	1 to 255 characters (default = 4)
LONG	1 to 255 characters (default = 8)
NUMERIC	1 to 28 characters
PACKED	1 to 14 characters (odd number of digits)
*PACKED	2 to 14 characters (even number of digits)

(Required parameter for BYTE, NUMERIC, PACKED, and *PACKED key types: defaults are provided for INTEGER, DOUBLE, REAL, and LONG key types, as noted above.)

keyblocking Number of keys per block. The *keyblocking* value is an even number greater than or equal to 4. It is used with the key entry size (*keysize* paramater) to determine the size of each key block according to the followign formula:

$$5 + \left(\frac{keysize+1}{2}\right) + 4)keyblocking = \text{key block size in words}$$

Five words are used for control information in each block, *keysize* specified in characters is divided by 2 to get the key size in words, and 4 words are added for the pointers in each key entry. This key entry size in words is multiplied by the *keyblocking* factor to determine key block size. If the *keyblocking* value generates a key block size greater than 2048 (2K) words, the file cannot be created.

The resulting key block size is rounded up to a multiple of 128 words. If the file has multiple keys, KSAM forces all key blocks to the same size and adjusts the number of keys per block accordingly.

Note that the value you specify for keyblocking may be increased (never decreased) by the system in order to produce a blocking factor that does not waste disc space. Refer to appendix B for a discussion of how the system determines the most efficient blocking factor based on the value you enter for *keyblocking*.

Key blocking can affect access time in that the smaller the key block, the more time it may take to retrieve a record using the key file. In many cases, the default blocking factor produces the most efficient key blocking.

(Optional parameter.)

Default: key blocking is set to a value that produces a key block size of 1024 (1K) words. (Maximum size is 2K.)

DUPLICATE
DUP

In order to allow duplicate key values, this word must be included in the KEY specification. If DUPLICATE (or DUP) is not specified, records with duplicate key values are rejected and an error message issued when such records are written to the file. DUP is a legal abbreviation of DUPLICATE. When you use this option to specify duplicate keys, each new duplicate key is inserted at the end of the duplicate key chain. This maintains the chronological order of duplicate keys.

RDUPLI-
CATE
RDUP

This option specifies that duplicate keys are allowed and are to be inserted randomly in the duplicate key chain. This method makes insertion of such keys faster, but does not maintain the chronological order of the duplicate key chain.

(Optional parameter.)

Default: If omitted, duplicate keys are prohibited.

KEYEN-
TRIES=
numentries

The value of *numentries* is used to determine the key file size. The value specified for *numentries* should be the maximum number of primary key entries expected. When there are alternate keys, KSAM automatically adjusts the key file size to accomodate each key in addition to the primary key.

Normally, this parameter can be omitted since KSAM assigns it the value of *numrec* (number of fixed-length data records or blocks of variable-length records). If, however, the data records are variable length and there are many small records, the value of *numrec* may be too small. In this case, you should specify a value for *numentries* greater than the value of *numrec*.

The number of key entries determines the size of the key file, the file limit. When a new KSAM file is created, the MPE end-of-file marker is set to this file limit rather than to the end-of-data as is normal for MPE files. This allows any key block to be accessed in case of system failure. To determine where the actual end-of-data is, use the KSAMUTIL VERIFY command, option 3, and look at the heading KEY FILE EOF. This shows the record number of the next available key block (one record past the last used key block).

(Optional parameter.)

Default: the value of numrec in the DISC= parameter or its default value 1024 if it too is omitted.

LABELS=
numlabels

The number of user label records to be created for the KSAM data file. Up to 254 labels (1 less than the MPE maximum) can be specified; COBOL programmers are restricted to 8 labels.

(Optional parameter.)

Default: if omitted, numlabels is equal 0.

KEY-
DEV=*device*

The device on which the key file resides, specified as a device class name or a logical device number. A device class name indicates the general type of the device as a string of one to eight alphanumeric characters beginning with a letter and terminated by a non-alphanumeric character such as a blank. The logical device number is the threecharacter numeric string identifying a particular device. If the data file is created on a remote device, the key file

is assigned to the same machine, and the key file device is specified in the KEYDEV= parameter.

Device class names and logical device numbers are assigned to devices during system configuration.

For KSAM files, the device must be a random-access device such as the disc.

(Optional parameter.)

Default: If omitted, the device class name DISC is used.

FIRSTREC= *recnum*
: Determines whether record numbers in the data file are to start with zero or one. If the integer 1 is specified, then records are numbered beginning with 1; otherwise they will start with 0. The only acceptable values for *recnum* are 1 and 0.

  Normally, record numbering in MPE files starts with zero, the default value for *recnum*. In order to be consistent with most commercial applications, you can specify FIRSTREC=1 to change the record numbering scheme so that data records are numbered starting with 1.

  (Optional parameter.)

  Default: if omitted, record numbering starts with zero.

LANG= *parameter*
: The name or identification number of the language to use in determining the key sequence. The language specified by *parameter* must be installed on the system. See Appendix F for detailed information on using Native Language Support (NLS) with KSAM files.

  (Optional parameter.)

  Default: if omitted, NATIVE-3000 is used.

NODUP LOCKS
: Prevents duplicate file locks ftom the same process. NODUPLOCKS controls "Lock" access to a KSAM file. Each time a process opens a file more than once and locks the file through a different open path, KSAM checks the file's lock access. If NODUPLOCKS was specified when the file was created, the process is not allowed to lock the file more than once. File locks from the same open path are always allowed.

  (Optional parameter)

  Default: if omitted, duplicate file locks from the same process are allowed.

KEY DESCRIPTION

Each key is described by specifying key type, key position, key size, and, optionally, the blocking factor and whether duplicates are allowed. Key type and size are defined in Table 2-2. Note that default values are provided for keysize when key type is specified as INTEGER, DOUBLE, REAL, or LONG. Only BYTE,INTEGER, and DOUBLE type keys can be used as generic keys.

Table 2-2. Key Types

keytype	*keysize* **(In Characters)**	**Format**
BYTE	1-255	Each character requires 8 bits of a computer word. A character may contain any of the HP ASCII character set consisting of letters of the alphabet, numbers, and special characters. (Refer to appendix C.)
INTEGER	1-255 (default = 2)	Single-word fixed-point format permits two's complement representation of positive and negative integers. Bit 0 is a sign bit and the remaining 15 bits define a quantity ranging from −32768 through +32767.
DOUBLE	1-255 (default = 4)	Double-word fixed-point format is the same as the integer format except that two words are linked together to allow a 32-bit quantity with a range between approximately −2 billion and +2 billion.
REAL	1-255 (default = 4)	Floating-point format with bit zero as a sign bit, an exponent (biased by +256) in bits 1 through 9, and a positive fraction in the remaining 22 bits of the double word. This type cannot be used as a generic key.
LONG	1-255 (default = 8)	Long floating-point format uses four words; an exponent (biased by +256) in bits 1-9, as with the real number, and a positive fraction in the remaining 54 bits. This type cannot be used as a generic key.
NUMERIC	1-28	External decimal format in which each decimal digit requires one 8-bit character and the sign is combined with the least significant digit. (Refer to Table 2-3 for the list of characters representing the digit/sign combinations.) This type cannot be used as a generic key.
PACKED	1-14	Packed decimal format in which each digit requires only 4 bits and the sign is specified as a hexadecimal number in the least significant 4 bits (1100 or C is plus and 1101 or D is minus). This type cannot be used as a generic key.
*PACKED	2-14	Same as PACKED except this key type contains an even number of digits. This type cannot be used as a generic key,

Table 2-3. Character Equivalent to Signed Digit for NUMERIC Keys

POSITIVE VALUES	POSITIVE VALUES	NEGATIVE VALUES	NEGATIVE VALUES
SIGNED DIGIT	CHARACTER	SIGNED DIGIT	CHARACTER
+0	{	-0	}
+1	A	-1	J
+2	B	-2	K
+3	C	-3	L
+4	D	-4	M
+5	E	-5	N
+6	F	-6	O
+7	G	-7	P
+8	H	-8	Q
+9	I	-9	R

CREATING A KSAM FILE

Creating a KSAM file with the KSAMUTIL BUILD command is very similar to creating a standard HP 3000 file with the MPE command :BUILD except that a KSAM file includes a key file description. As with standard files, the default values can be assumed for many of the file description parameters.

To create a KSAM file from the KSAMUTIL program, you can start by simply naming the file as the first parameter of the BUILD command. The file name defines the data file portion of the KSAM file with the default options: fixed-length, 128-word, binary-coded records, blocked 1 record per block.

To fully define a KSAM file, you must also:

- name the key file
- define at least one key (the primary key) in terms of:

  type
  location in the data file
  size

These parameters provide your minimum KSAM file description from which the file can be created. To illustrate:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD KSAMFILE;KEYENTITY =KFILE;KEY=I,21,2
```

This command assigns the name KSAMFILE to the KSAM data file; it names the key file KFILE, and defines the primary key as an integer that starts in character 21 of the record, and is two characters long. By default, the blocking factor of the keyfile provides key blocks 1024 words long, the maximum number of primary keys is set to 1023 (the same as the

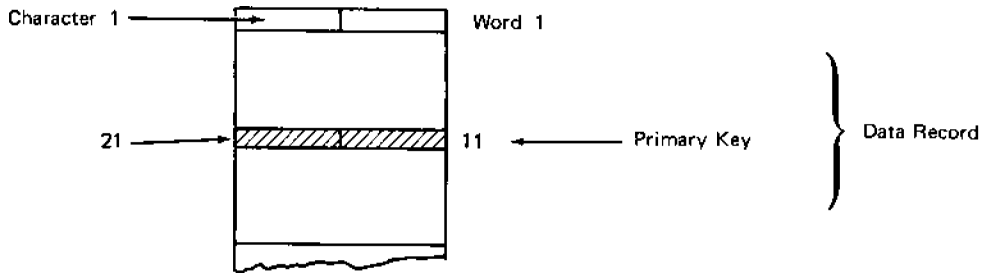
maximum number of data records), duplicate keys are prohibited, and record numbering starts with zero.

File KSAMFILE is now created. Default values were used where possible so that the BUILD command specification shown above is the minimum needed to create a KSAM file. You could create the same file, KSAMFILE, with the following BUILD command in which default parameters are specified.

```
                    recsize
                      |
>BUILD KSAMFILE;REC=128,,F,BINARY& <------ line continuation character
>              ;DEV=DISC&
>              ;CODE=0&
>              ;LABELS=0&
>              ;FIRSTREC=0>              ;DISC=1023,8,1&  <------numrec,numextents,initialloc

>     ;KEYFILE=KFILE&
>              ;KEY= I,21,2& <------ key description
>              ;KEYENTRIES=1023&   <-----    numentries
>              ;KEYDEV=DISC
```

This specification of the BUILD command, although initially more cumbersome, documents the default values with which the file is created. Since the default keyblocking factor is a value calculated from the key size so that each key block is 1K words long, it is not specified here. You can use the VERIFY command to find the value KSAM has assigned as a key blocking factor for any file you create using a default for this value.

Only a primary key is defined for this file. Within the data file, this key is an integer that occupies characters 21 and 22 (word 11) of each data record.



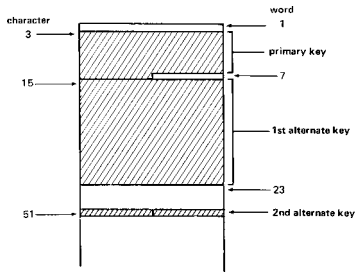
In the key file, the values in any key are ordered sequentially so that the next higher value can always be located. The key should not begin in the first two characters of the data record since these characters are set to all 1's when the record is deleted. If the key value of deleted records need never be recovered, then this restriction can be ignored.

For each alternate key in addition to the primary key, another KEY= clause must be included. Suppose a personnel file with a primary key containing an employee number, an alternate key containing a name, and another alternate key containing the person's age. The first two keys are specified as BYTE keys, the third is an INTEGER. The key file is blocked with 10 keys per block and the maximum number of primary keys expected is 3000:

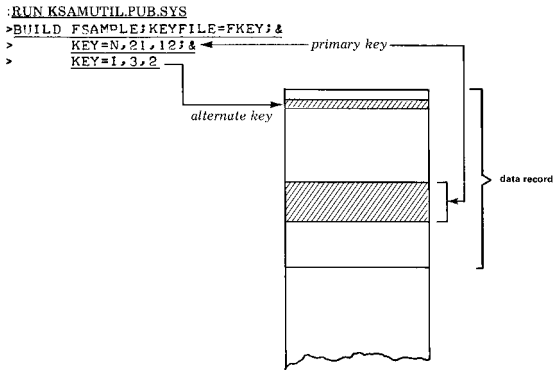
```
:RUN KSAMUTIL.PUB.SYS
>BUILD EMPLOYEE;REC=,,,ASCII;KEYENTITY =EMPKEY;KEYENTRIES=3000;&
>      KEY=B,3,11,10;&<---------- primary key (employee number)
```

```
>       KEY=B,15,30,10;&<---------  alternate key (employee name)
>       KEY=I,51,2,10&<-----------  alternate key (employee age)
```

The keys are located in the data record as follows:



Note that the keys need not be contiguous. In this example, the primary key is located nearer to the beginning of the record than the other keys. This is not a requirement; the primary key can physically follow any alternate keys in the record, although the primary key is always the first key specified in the BUILD command. For example, in the file FSAMPLE, the primary key starts in character 21 following a secondary key in character 3:



ERASE

Clears the contents of a KSAM file.

```
>ERASE filereference
```

The contents of a KSAM file, both the data and key files, can be cleared to an empty state with the KSAMUTIL ERASE command.

PARAMETERS

filereference Actual file designator that identifies the KSAM data file. It is specified exactly like *filereference1* in the >BUILD command.

(Required parameter.)

CLEARING A KSAMFILE

EXECUTE access is required to erase a KSAM file. If the group in which the KSAM file resides does not have EXECUTE access, any attempt to erase a file results in an error and the message, "FSERR 93 SECURITY VIOLATION."

When ERASE is specified for a KSAM file, the end-of-file pointer that follows all data is reset to point to the first record in the data file. This position of the pointer is identical to its position when the file is created and before any data is written to the file.

All pointers and control words in the key file are reset to indicate that the data file is empty.

Note that the file is still created and new data may be written to it.

For example, to clear the contents from the file identified as KSAMFILE:

```
>ERASE KSAMFILE
```

PURGE

Purges a KSAM file from the system.

```
>PURGE filereference [,TEMP]
```

The KSAMUTIL PURGE command can be used to remove a KSAM file, both data and key files, from the system. Although the MPE :PURGE command can also be used, it must be specified twice, once for the data file and once for the key file. If you are programming in COBOL, BASIC, or RPG, you should use the KSAMUTIL PURGE command to purge a KSAM file. In SPL or FORTRAN you could also use the FCLOSE intrinsic (described in section IV) to purge a KSAM file.

PARAMETERS

filereference Actual file designator identifying the KSAM data file. Specified exactly like *filereference1* in the >BUILD command.

(Required parameter.)

TEMP Must be specified if file is a temporary file in session/job temporary file domain. If omitted, a permanent file is assumed.

(Optional parameter.)

PURGING A KSAM FILE

When PURGE is executed, the specified KSAM data file and its associated key file are removed from the system and can no longer be referenced.

For example, to purge a temporary KSAM file called KTEMP:

```
>PURGE KTEMP,TEMP
KTEMP.KSAM.DATAMGT & KKEY PURGED.
```

To purge the permanent file KSAMFILE:

```
>PURGE KSAMFILE
KSAMFILE.KSAM.DATAMGT & KFILE PURGED.
```

The system prints the data and key file names of a successfully purged KSAM file. It also prints the group and account names in which the file was created (in this case KSAM and DATAMGT).

RENAME

Renames either the data or key file of a KSAM file.

> { RENAME / R } *oldfilereference, newfilereference* [,TEMP]

The KSAMUTIL RENAME command can be used to change either the KSAM data file name or the KSAM key file name to a new name. Following execution of RENAME, the data and key files retain their relation to each other. *Note that if the MPE :RENAME command is used, this relation is severed.* The FRENAME intrinsic cannot be used to rename a KSAM file.

PARAMETERS

oldfile-reference	Current actual file designator identifying the KSAM data file or the KSAM key file, specified exactly like *filereference1* or *filereference2* in the BUILD command. (Required parameter.)
newfile-reference	New actual file designator in same format as *oldfilereference*. The file named by *oldfilereference* will be given the name specified by *newfilereference*. (Required parameter.)
TEMP	Indicates that old file was, and new file will be, a temporary file in the session/job temporary file domain. (Optional parameter.) Default: If omitted, permanent file is assumed.

RENAMING A KSAM FILE

You may rename either the data file or the key file, not both, with one >RENAME command. To rename the entire file, you must specify the RENAME command twice. Thus, to rename the data file KSAMFILE and its associated key file KFILE:

```
>RENAME KSAMFILE,NEWDATA
>RENAME KFILE, NEWKEY
```

The relation between keys and data in the newly named files is the same as that in the files KSAMFILE and KFILE.

If the data file being renamed was protected by a lockword, then this lockword must be specified on both the old and new files if it is to be retained. If the lockword is omitted, it is removed when the file is renamed. Note that a lockword is never specified when renaming the key file; the keyfile is protected automatically by any lockword assigned to its associated data file. For example, to assign a new lockword to the data file DATAFIL:

```
>RENAME DATAFIL/LOCKA, DATAFIL/LOCKB <----------- new lockword
```

Note that the new file name need not be in the same group as the old file name. RENAME provides a way to move a file from one group to another. For example, to move the KSAM file DATAFILE with its associated key file KEYFILE from GROUPA to GROUPB:

```
>RENAME DATAFILE.GROUPA,DATAFILE.GROUPB
```

Note that only one RENAME command is used. This one command insures that both the data file and the key file are in the same group.

SAVE

Saves a temporary KSAM file as a permanent file.

$$>\left\{\begin{array}{l}\texttt{SAVE}\\ \texttt{S}\end{array}\right\}[\textit{filereference}]$$

A temporary KSAM data file and its associated key file are made permanent with the KSAMUTIL SAVE command. The keyword "SAVE" can be abbreviated to "S".

PARAMETERS

filereference Actual file designator identifying the session/job temporary file to be saved, specified exactly like *filereference1* in the >BUILD command.

(Optional parameter.) *Default: If omitted, last filereference is assumed.*

SAVING A KSAM FILE

Assume that KSAM data file KDATA and its associated key file was created as a session/job temporary file; to save this file as a permanent file:

```
>SAVE KDATA
```

Both the data and key files are saved.

VERIFY

Displays access and status information about KSAM file.

```
>{ VERIFY } [filereference] [;OFFLINE] [;NOCHECK]
 { V      }
```

With the VERIFY command, you can request a display of the characteristics of a KSAM data file, both the static information defined at file creation and dynamic file access information. The abbreviation V can be used instead of the keyword VERIFY.

PARAMETERS

filereference Actual file designator identifying the file whose characteristics are to be displayed. The actual designator can be a back reference to a file name in an MPE :FILE command; in this case, the actual designator must be preceded by an asterisk (*). Either the data file name or the key file name may be used to identify the KSAM file.

(Optional parameter only if no parameters.)

Default: If omitted, last filereference is assumed.

OFFLINE Display output on line printer. An MPE :FILE command may be used to specify a particular line printer.

(Optional parameter.)

Default: If omitted, display is sent to terminal.

NOCHECK Allows specified KSAM file to be opened for read-only access by the VERIFY command; use when a system failure prevents the KSAM file from being opened.

(Optional parameter.)

Default: If omitted, VERIFY cannot open file that was open when system failed.

DISPLAY KSAM FILE CHARACTERISTICS

In a session, you will be asked to select one of four possible displays:

1. File information (definitions from file creation plus file access statistics)
2. KSAM parameters (definitions of keys from file creation)
3. KSAM control (key file access statistics)
4. All three of the above displays

In a job, the entire set of displays is printed exactly as if option 4 had been selected in a session.

To illustrate the interaction, the following VERIFY commands select each of the three separate displays; if option 4 were selected, these displays would be printed consecutively with no halt until they were finished. User entries are underlined:

```
>RUN KSAMUTIL.PUB.SYS
```

```
 HP32208A.2,4  TUE, APR 17, 1979, 11:23 AM      KSAMUTIL VERSION:A.2.4
>VERIFY TESTFILE

                select file information only-------------------------|
WHICH  (1=FILE INFO, 2=KSAM PARAMETERS,  3=KSAM CONTROL, 4=ALL)?1

TESTFILE,JOAN.MORRIS        CREATOR=JOAN
FOPTIONS(004005)=KSAM, :FILE, NOCCTL, F, FILENAME, ASCII, PERM
AOPTIONS(000400)=DEFAULT, NOBUF, DFFAULT, NO FLOCK, NO MR, IN
RECSIZE:SUB:TYP:LDNUM:DRT:UN.:  CODE:LOGICAL PTR: END OF FILE:FILE LIMIT
   -128:  9:  0:    2:  4:  1:     0:          0:           5:      1023
 LOG. COUNT:PHYS. COUNT:BLK SZ:EXT SZ:NR EXT: LABELS:LDN:   DISCADDR:
         1:          1:  -128:   129:     8:      0:  2:00000117760:
```

The infomation returned by selecting file information is the same as that returned by FGETINFO (described in section IV).

```
                select key file information only-------------------------|
                                                                         |
WHICH (1=FILE INFO,   2=KSAM PARAMETERS,   3=KSAM CONTROL,   4=ALL)?2

KEY FILE=TESTKEY  KEY FILE DEVICE=3            SIZE=         386  KEYS=   2
FLAGWORD(000000)=RANDOM PRIMARY,    FIRST RECORD=0,   PERMANENT
KEY TY LENGTH   LOC. D KEY BF  LEVEL
  1  B     20      3 Y     72      1<----- primary key is listed first,
  2  B      8     24 R    126      1<----- alternate keys follow
     \---------------------/
                    ^
                    |
                    |----------------------- random insertion of duplicate key
   corresponds to KEY= descriptions in BUILD command
```

The actual number of keys per block (the blocking factor) is listed in this display under the heading KEY BF. Note that this number may be greater than the blocking factor you specified during file creation. This occurs if KSAM adjusts the specified blocking factor to generate a block size that makes optimum use of disc space, KSAM only increases the specified blocking factor, it never decreases it. (Refer to appendix B for full particulars on the calculation of block size and the adjustment of the blocking factor.)

The maximum number of levels in the key file structure for each key is noted under the heading LEVEL.

```
                select dynamic KSAM file information------------|
                                                                |
WHICH (1=FILE INFO,  2=KSAM PARAMETERS,  3=KSAM CONTROL,  4=ALL)?3

DATA FILE = TESTFILE     VERSION= A.2.4
KEY CREATED=107/'79  10:58:17.4      KEY ACCESS= 107/'79 11:23:35.9
KEY CHANGED=107/'79  11:21: 7.7      COUNT START=107/'79 11:21: 8.0
DATA RECS  =           5 DATA BLOCKS=          4 END BLK WDS=         64
DATA BLK SZ=          64 DATA REC SZ=        128 ACCESSORS=           0
FOPEN                  1 FREAD                 0 FCLOSE                1
FREADDIR               0 FREADC                0 FRFADBYKEY            0
FREMOVE                0 FSPACF                0 FFINDBYKEY            0
FGETINFO               3 FGETKEYINFO           1 FREADLABEL            0
FWRITELABEL            0 FCHECK                0 FFINDN                0
FWRITE                 5 FUPDATE               0 FPOINT                0
FLOCK                  0 FUNLOCK               0 FCONTROL              0
FSETMODE               0
KEYBLK READ            3 KEYBLK WROTE          2 KEYBLK SPLIT          0
KEY FILE EOF          18 FREE KEY HD           0 SYSTEM FAILURE        0
```

```
MIN PRIME              3 MAX PRIME             1 RESET DATE
DATA FIXED          TRUE DATA B/F              1 TOTAL KFYS            2
FIRST RECNUM           0 MIN PECSIZE          31
```

The dynamic key file information displayed by option 3 together with the static key file information displayed by option 2 comprise the information displayed by the FGETINFO intrinsic described in section IV. Note that the version number displayed by VERIFY is the version of KSAM under which the file was created. The intrinsic HP32208 described in section IV can be used to determine the current version of KSAM.

TERMINATING THE >VERIFY COMMAND

In order to terminate the VERIFY command in a session, you must press the RETURN key (CR) in response prompt:

```
WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?
```

Any other response either causes a display followed by reiteration of this prompt or else causes this prompt to be issued. For example:

```
WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?6

WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?0

WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?CR
                                                                |
>EXIT                                                           |
                                                                |
 END OF PROGRAM     only response to terminate VERIFY.----------|
```

DIRECTING VERIFY OUTPUT TO LINE PRINTER

If you want a "hard copy" of the information displayed by VERIFY, you should use the OFFLINE option. When included, OFFLINE directs the VERIFY information to a line printer. You can use the :FILE command to specify a different output device or a particular line printer.

USING VERIFY FOR RECOVERY

In case of a system failure when a KSAM file is open, the VERIFY command provides the following useful information:

- EOF marker on the data file (END OF FILE header in option 1)
- EOF marker on key file (KEY FILE EOF header in option 3)
- Flag indicating whether system failure occurred (SYSTEM FAILURE heading in option 3)
- Number of processes that had opened the file for non read-only access when the system failure occurred (ACCESSORS heading in option 3).

Note that you must use the NOCHECK option in order to run VERIFY when a system failure prevents KSAM files from being opened. This option overrides the restriction on opening files after a system failure and allows a file to be opened for read-only access in order to get the VERIFY information. (This access is not counted in the ACCESSORS count.)

(Refer to the KEYINFO command description for an example of how the VERIFY information can be used for recovery after system failure.)

KEYSEQ

Verifies sequence of key values in KSAM file.

```
>{ KEYSEQ } [filereference] [;SEQ=keysequence] [;OFFLINE] [;NOLIST]
 { KS     }
```

This command compares all the key values in a particular key (primary or alternate) to determine whether they are in ascending sequence. If any values are out of sequence, a list of numbers identifying such values is displayed, unless NOLIST is specified. In any case, the number of outquence values is returned. Note that if key values are out of sequence, the key file is damaged and the KSAM file must be reloaded.

The abbreviation KS may be used instead of the keyword KEYSEQ.

PARAMETERS

filereference Actual file designator identifying the KSAM file whose key values are to be verified. Either the data file name or the key file name can be used to identify a KSAM file. Also, a back reference to a file named in an MPE :FILE command may be used.

(Optional parameter if no parameters are specified.)

Default: If omitted, the last file referenced is assumed.

SEQ=*keysequence* Identifies particular key whose key values are to be checked. Keys are numbered from 1. The first key (SEQ=1) is always the primary key; subsequent keys are alternate keys numbered in the order they appear in the record, such that the first alternate key in the record is SEQ=2, the second alternate key is SEQ=3, and so forth.

(Optional parameter.)

Default: If omitted, the primary key is assumed.

OFFLINE Directs list of out-of-sequence keys to the line printer. An MPE :FILE command may be used to indicate a different output device than the line printer, or a particular line printer.

(Optional parameter.)

Default: If omitted, the list of out-of-sequence keys is displayed at user's terminal.

NOLIST Suppresses display of the particular key numbers whose values are out of sequence. A count of the out-of-sequence values is displayed even if NOLIST is specified.

(Optional parameter.)

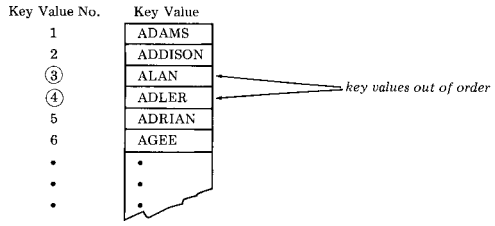
Default: If omitted, list of key numbers for out-of-sequence key ualues is displayed.

VERIFY KEY SEQUENCE

If you suspect that your key file has out of order key values in any key, you can run KEYSEQ. If any key values are not in ascending sequence, the key numbers associated with those key values are displayed. Note that the number of a key value refers to its position in the key file. For example, if the third, fourth, and fifth key values of a particular key are out of sequence, the numbers 3, 4, and 5 are displayed.

If the list of out-of-sequence key numbers is long, you can terminate it by holding down the CNTL key while typing Y (CNTL/Y). The total number of out-of-sequence key values will be displayed even if you terminate the list with CNTL/Y or suppress it altogether with NOLIST.

Consider the following partiallist of key values in the primary key of the KSAM file MYFILE:



If you run KSAMUTIL and use the KEYSEQ command, as shown below, you can determine which keys are out of order:

```
:RUN KSAMUTIL.PUB.SYS
>KEYSEQ MYFILE ;SEQ=1<-----------  test primary key sequence
```

KEYSEQ displays the following:

```
KEY VALUE # (FOR VALUE OUT OF SEQUENCE)
3 ------  key number of keys with out-of-sequence values
4 -------|
TOTAL # OF KEY VALUES READ       30
# OF KEY VALUES OUT OF KEY SEQUENCE ORDER            2
KEY FILE STRUCTURE DAMAGED, KSAM FILE HAS TO BE RELOADED
```

Regardless of the number of key values that are out of sequence, you should reload the KSAM file to restore its integrity.

Consider a second example. Suppose MYFILE has two alternate keys, one starting in location 11 (11th character of record) and another starting in location 33. To verify the sequence of key values in the second alternate key, execute KEYSEQ as follows:

```
>KS MYFILE ;SEQ=3 ;NOLIST <-----  suppress list of key numbers
                |
                |------  second alternate key

TOTAL # OF KEY VALUES READ 30
#OF KEY VALUES OUT OF KEY SEQUENCE ORDER 0 <---  key values in sequence
```

In this case all the key values were in correct sequence. Unless other keys in this file have values that are out of sequence, you need not reload the file.

Note that NOLIST was specified. In general this is good practice. If any key values are not in sequence, the file should be reloaded, so it is seldom important to know which keys are out of sequence.

KEYDUMP

Provides formatted, structural dump of key file.

```
>{ KEYDUMP } [filereference] [;SEQ=keysequence]
 { KD      }
[;SUBSET= { [-]position "string" } [,number] ]  [;FILE =formaldesignator]
[;OFFLINE] [;SORT]
```

The key file dump consists of three items of information for each key value:

1. Key Value	The actual value of each key in ascending order
2. Record Pointer	The record number (fixed-length files) or word offset (variable-length files) of the data record to which the associated key value points.
3. Key Block Address	Relative record number of the first record in the key block containing the associated key value, followed within parentheses by the number of key values in the block. The addresses of key blocks at different levels are indented.

This dump is very useful for examining the contents of any key file. Since key blocks are physically scattered throughout the key file, linked by pointers, it is difficult to follow an unstructured dump of a key file. The KEYDUMP display shows the contents of the key file, not as they are actually stored, but in a way that makes it much simpler to read than a dump of the actual file.

One key at a time is dumped by KEYDUMP. If there is more than a primary key, you must run KEYDUMP for each key in order to dump the entire key file.

Note that you can use CNTL/Y (the CNTL key held down while pressing the Y key) to stop display of this dump at any time. This is particularly useful if you display the dump at the terminal. Usually, however, you will use the OFFLINE option to list the dump on a line printer (see Parameter description, below).

You can choose to dump a subset of the key file contents based on the key number or a key value. You can send the dump to a particular file and, if so, you can sort the key file contents by the record number in the data file rather than by key value. You can also send the dump to a line printer.

PARAMETERS

filereference	Actual file designator identifying the KSAM file whose key file is to be dumped; either the data file name or the key file name can be specified. The *filereference* can be a back reference to a file named in an MPE :FILE command. (Optional parameter if no parameters are specified.) *Default: If omitted, the last file referenced is assumed.*
SEQ= *keysequence*	Specify a particular key whose contents are to be dumped. The primary key, whatever its location in the data record, is always key number 1 (SEQ=1). Alternate keys are numbered according to the order in which they are specified in the BUILD command (or in *ksamparam* at FOPEN). The first alternate key is specified as SEQ=2, the next alternate key as SEQ=3, and so forth.

(Optional parameter.)

Default: If omitted, the primary key is selected.

SUBSET= Select a portion of the key file to dump, based on the numeric position of the key or the key value, and the number of key values.

[−]*position* Start dump with key whose number is specified. This number is the same as the key number issued by KEYSEQ. It corresponds to the position of the key value in the file in ascending sequence. Thus the first key value is *position* 1, the second is *position* 2, and so forth.

The optional minus sign suppresses the normal indentation by key levels of the key block address display.

"*string*" Start dump with first key value greater than or equal to the specified string.

,*number* Indicates the number of key values to be dumped starting with the key at the indicated *position* or whose value is indicated by "*string*".

(Optional parameter.)

Default: If omitted, all the key ualues for selected keys are dumped.

FILE =*filename* Direct key file dump to specified disc file. A disc file (*filename*) will be created with a record size equal to the size of a key entry, that is keylength (rounded up to full words) + four words.

The four words are needed for the record pointer (2 words) plus the key block address (2 words). Note that a new file is always created, so do not name an existing file.

The file has a default block size of 1K words. Any of the file characteristics *except the record size* can be changed by a :FILE statement. For example:

```
>:FILE FILEDUMP ;REC=,100 ;DEV=TAPE
>KEYDUMP MYFILE ;ENTITY =*FILEDUMP
```

These commands dump the primary key sequence to a tape with 100 records per block.

(Optional parameter.)

Default: If omitted, key dump is sent to terminal.

OFFLINE Direct output to a line printer. An MPE :FILE command can be used to indicate a different output device or a particular line printer.

(Optional parameter.)

Default: If omitted, the dump is sent to the user's terminal.

SORT Sort dump by record pointers rather than key values. The record pointers indicate the record number of the records in a data file with fixed length records or the word offset of the records in a data file with variable-length records.

Note that this option can be used only when the dump is directed to a specific file with the ENTITY = option.

(Optional parameter.)

Default: If omitted, key dump is in ascending sequence by key value.

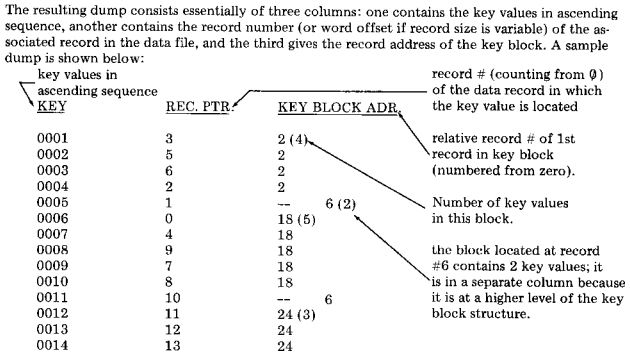
DUMPING THE KEY FILE

The dump produced by KEYDUMP consists of three columns: the first contains the key value, the second a pointer to a record number in the data file, and the third contains the key block address and the number of key values in that block. The key block address is given as the record number of the first record in any block.

For example, assume that TESTFILE contains an INTEGER type primary key whose values we want to see. Run KEYDUMP as follows:

```
:RUN KSAMUTIL.PUB.SYS
>KEYDUMP TESTFILE
```

The resulting dump consists essentially of three columns: one contains the key values in ascending sequence, another contains the record number (or word offset if record size is variable) of the associated record in the data file, and the third gives the record address of the key block. A sample dump is shown below:



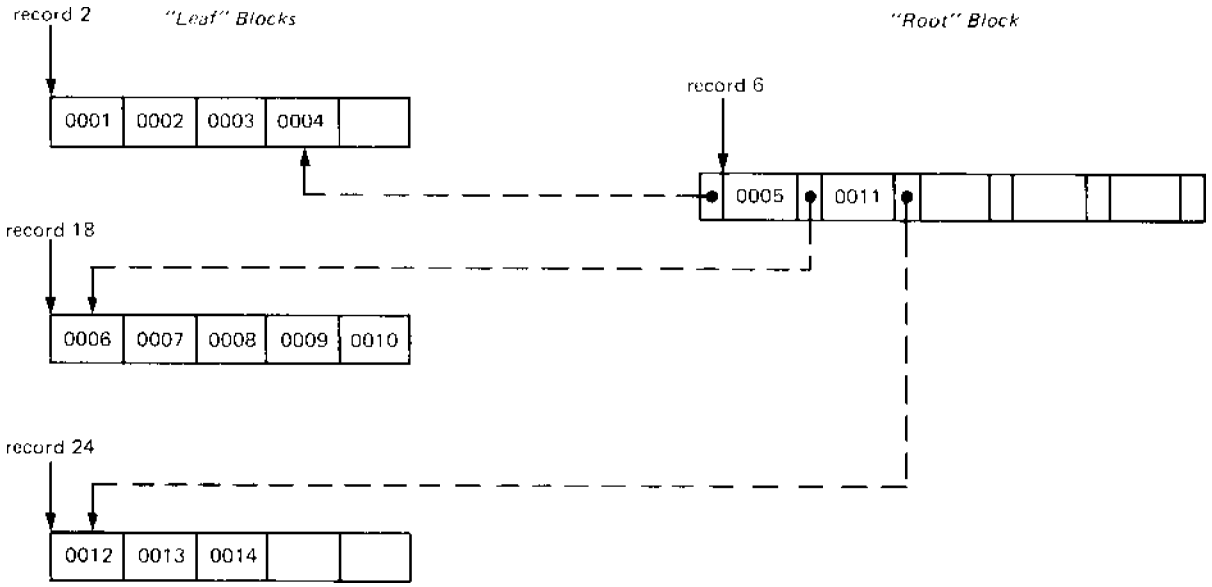
This dump lists under the heading "KEY" 14 integer key values in ascending order from 0001 through 0014. The next column under "REC.PTR." lists the record number of the data record associated with the key value—thus, key value 0001 is in record number 3 of the data file (the fourth record in chronological sequence), and key value 0006 is in the first record in the data file, record number 0. The third column under "KEY BLOCK ADR." shows the address of the key block in which each key value resides, The key block address is shown as the record number of the first record in the key block. (Note that KSAM key files use fixed

length records each one sector long—128 words. Thus, the record number is also the sector number. A keyblock consists of more than one sectors—default is 8 sectors).

Key values are organized into blocks using a B-Tree structure (refer to appendix B for details). This structure has one or more levels where the first or highest level, is known as the "root" and lower level are "leaves". This dump shows the level structure of the key file by indenting the key block addresses to correspond to levels. The highest or root level address is in the rightmost column, lower levels are listed to the left. By looking at the key block address, we see that the key block starting at record (sector) 6 is the root block, and that there are three key blocks at a lower level whose addresses start, respectively, at records 2, 18, and 24. This key file has two levels; a key file with more levels would have correspondingly more columns under the key block address heading.

The first time a key block address is listed, it is followed in parentheses by the number of key values in that block. Looking at the dump, we see that the block starting at record 2 has 4 key values, the block at record 6 has 2 values, the block at record 18 has 5 values, and the block at record 24 has 3.

With this dump, we can picture the structure of the key file associated with the file TESTDUMP:



The key values are shown within their key blocks; the dashed lines show the pointers that link key blocks in ascending sequence.

DUMPING A SUBSET OF THE KEY FILE

If you want to dump a selected number of key values rather than all the values in a key, you can use the SUBSET option of KEYDUMP. The starting key value can be located in two ways: if you know the key number of the first key value you want displayed, use the SUBSET=*position* format; if you know the actual key value (or a value less than the key value), then you can use the SUBSET=*"string"* format. In either case, the second SUBSET parameter is always an integer that indicates the number of key values you want dumped.

BY KEY NUMBER

The key number is the sequential number associated with each key value in a particular key. If the KEYSEQ command has listed key numbers that are out of sequence, you may want to dump only these values. Suppose that TESTFILE has out of sequence values, the following example runs KEYSEQ first and then runs KEYDUMP to dump the key values shown as out of sequence. (ln order to see the last value in correct sequence, the key preceding the first key out of sequence is selected as the first key to dump.)

```
>KS TESTFILE
KEY VALUE # (FOR KEY VALUE OUT OF SEQUENCE)
6<-------------------------|
7                          |
8                          |values used for SUBSET= of KEYDUMP
9                            |------------|
TOTAL # OF KEY VALUES READ 14             |
# OF KEY VALUES OUT OF KEY SEQUENCE ORDER 4 -- values used for SUBSET= of KEYDUMP
KEY FILE STRUCTURE DAMAGED, KSAM FILE HAS TO BE RELOADED
>KD TESTFILE ;SUBSET=5,5 <------- number of key ualues out of sequence (Plus 1)
                     |
                     |----- key preceding first key # out of sequence
```

The following dump shows the last key value in sequence followed by the key values that are out of sequence:

```
                 KEY            REC.PTR.             KEY BLOCK ADR.

1st key out    0005             3                           12(2)
of sequence--->0008             9                   18 (5)
               0007             4                   18
               0009             7                   18
               0006             0                   18
```

BY KEY VALUE

The second version of SUBSET= specifies an actual key value followed by the number of key values. You need not specify the exact key value; it can be a value less than an actual Integer or Double type key value (approximate match) or only the first part of a Byte type key value (generic match). For example, suppose TESTFILE has an alternate key that contains names in alphabetic order and you want to look at the ten key values that start "GI" or the next greater value. Specify the following command:

```
>KD TESTFILE ;SEQ=2 ;SUBSET="GI",10
                  |          \____/----
                  |                |
                  |                ---------- dump 1st 10 values
                  |                             starting with or
                  |                           greater than "GI"
                   ----- 1st alternate key
```

The dump appears as follows:

```
KEY          REC.PTR.          KEY BLOCK ADR.

GIBBS        3                 2 (4)
GILLESPIE    12                2
GLADSTONE    4                 2
HERTZ        8                 2
HIGGINS      0                 -     8(3)
JONES        7                 16(4)
LOOMIS       13                16
MORRIS       5                 16
MYERS        6                 16
NOLAN        1                 -     8
```

SORTING DUMP BY RECORD POINTER

If you use the SORT option of KEYDUMP, you must also specify ENTITY =*filename*, where the specified file name is that of a disc file. (Note that you must not name an existing file; a new file is created for the dump.) In this case, you might also want to suppress the indention of the key block address levels. To do this, enter the following command:

```
                                  |------suppress key block address indentation
                                  |
                                  |
>KD TESTFILE;FILE =MYFILE;SUBSET=-1,500;SORT <--- sort by record pointer
                   |
                   |---- dump is sent to MYFILE, created with default values
```

The resulting dump is sent to a disc file MYFILE, created with a default block size of 10 words, one record per key entry. The key entries are sorted by the pointers to the records in the data file. Indentation of the key block address is suppressed. The key values, record pointers, and key block addresses are not converted to ASCII but are dumped to the specified file in binary format. In case of a file with 500 or fewer key values, the entire file is dumped.

The SORT option is useful if you want to look at key values in terms of the data records to which the key values point. For example, in order to determine whether any key values are missing, you can dump all the keys in a file using the SORT option, and compare the record numbers in each dump to make sure each record has the same number of key values pointing to it.

KEYINFO

Displays information about the key file, and attempts recovery of a KSAM file in case of system failure when the file is open.

```
>{ KEYINFO } [filereference] [;OFFLINE] [;RECOVER]
 { K1      }
```

KEYINFO performs two operations: it collects and displays information about the key file, and it takes steps to recover the KSAM file in case a system crash occurred when the file was open. The second operation is performed only after a system crash or if the RECOVER parameter is specified.

The key information displayed by KEYINFO consists of:

- Number of levels in key block structure:
- Number of key blocks
- Number of sectors per key block
- Number of keys in root block
- Number of keys in all blocks of the key file
- Percent of each key block used
- Largest key block address

The crash recovery performed by KEYINFO depends on the type of damage to the file.

- If MPE end-of-file does not match end-of-file for KSAM data file, KEYINFO resets the MPE end-of-file to match the KSAM end-of-file.
- If key file contains values that point to records past the KSAM end-of-file, KEYINFO deletes these key values.
- If the key file end-of-file marker does not match the actual end of the key file, KEYINFO corrects the key file end-of-file marker.
- If records in the data file do not have associated key values in the key file, KEYINFO issues a warning that key values are missing.

PARAMETERS

filereference Actual file designator of the KSAM file; either the data file name or the key file name may be specified. The filereference can be a back reference to a file named in an MPE :FILE command.

(Optional parameter if no parameters are specified.)

Default: If omitted, the last file referenced is assumed.

OFFLINE Directs output to the line printer. An MPE :FILE command can be used to indicate a different output device or a particular line printer. (Optional parameter.)

Default: If omitted, output is sent to user's terminal.

RECOVER Forces KEYINFO to perform recovery procedures even though no system crash occurred.

(Optional parameter.)

Default: If omitted, recovery performed only if system crashed with file open.

REQUESTING KEY FILE INFORMATION

Information is displayed by KEYINFO for each key in the key file, in key order starting with the primary key. For example, request KEYINFO for the file DATAFIL which has three keys:

```
>KI DATAFIL

--------- INFO FOR KEY               1 ---------

# OF LEVELS OF B-TREE                        1
# OF KEY BLOCKS                             1
# OF SECTORS PER KEY BLOCK                  8
# OF KEYS IN ROOT KEY BLOCK                20
# OF KEYS IN B-TREE                        20
% OF KEY BLOCK UTILIZATION                 38.4
THE LARGEST KEY BLOCK ADDRESS               2

--------- INFO FOR KEY               2 ---------

# OF LEVELS OF B-TREE                        1
# OF KEY BLOCKS                             1
# OF SECTORS PER KEY BLOCK                  8
# OF KEYS IN ROOT KEY BLOCK                20
# OF KEYS IN B-TREE                        20
% OF KEY BLOCK UTILTZATION                  9.9
THE LARGEST KEY BLOCK ADDRESS               10

--------- INFO FOR KEY               3 ---------

# OF LEVELS OF B-TREE                        1
# OF KEY BLOCKS                             1
# OF SECTORS PER KEY BLOCK                  8
# OF KEYS IN ROOT KEY BLOCK                20
# OF KEYS IN B-TREE                        20
% OF KEY BLOCK UTILIZATION                 13.8
THE LARGEST KEY BLOCK ADDPESS               18

>EXIT
```

 # OF LEVELS OF B-TREE - Key files are organized in a structure known as a "B-Tree". This structure may have one or more levels (for details refer to appendix B). *The file DATAFIL has only one level.*

 # OF KEY BLOCKS - Key values are stored in blocks; this entry gives the total number of key blocks in the file. *DATAFIL has only one key block.*

 # OF SECTORS PER KEY BLOCK - A key block may require one or more 128-word sectors. *DATAFIL uses eight sectors for its key block (the default value).*

 # OF KEYS IN ROOT BLOCK - This specifies the number of key values stored in the root block (in this case the only block). If this number is equal to the key blocking factor (see KEY BF header in VERIFY output), then the next key block split will increase the number of levels in the B-Tree by one. *DATAFIL has 20 key values in its root block, and the blocking factor allows 52, 202, or 144 (see VERIFY printout below).*

```
>VERIFY DATAFIL

WHICH  (1=FILE  INFO, 2=KSAM PARAMETERS,  3=KSAM CONTROL, 4=ALL)?2
```

```
KEY ENTITY=KEYFIL   KEY FILE DEVICE=2         SIZE=        50   KEYS=   3
FLAGWORD(000000)=RANDOM   PRIMARY, FIRST RECORD=0, PERMANENT
KEY TY LENGTH   LOC. D   KEY BF  LEVEL
                              |
                              |----------- blocking factor, keys/block
  1  B     30      1 Y       52      1
  2  B      2     31 Y      202      1
  3  B      6     33 Y      144      1
```

 # OF KEYS IN B-TREE - This is the total number of key values in the key file for each key. This number should be the same for each key and should also be the same as the number of active records in the data file (to determine this, use the FCOPY command >FROM=DATAFIL;TO=$NULL ;KEY=O. FCOPY is described later in this section). *DATAFIL has 20 key values in each B-Tree, and this is the same number as the number of active data records (see FCOPY output below).*

```
:RUN FCOPY.PUB.SYS

HP32212A.3.08 FILE COPTER (C) HEWLETT-PACKARD CO. 1978

>FROM=DATAFIL;TO=$NULL;KEY=0
EOF FOUND IN FROMFILE AFTER RECORD 19 <---- records numbered from 0

20 RECORDS PROCESSED *** 0 ERRORS
```

% OF KEY BLOCK UTILIZATION - Average percent of use of all key blocks (percent of use means how much of the block contains key values). Note that the root block of a multi-level tree is omitted from this average. For multi-level trees the percent is between 50% and 100%, for single-level trees between 0% and 100%. The higher the percentage, the faster the retrieval of data. But, also the higher the percentage, the greater the chance of block splits when records are added. *DATAFIL uses 38.4% for its primary hey, 9.9% and 13.8% each for its two alternate keys.*

THE LARGEST KEY BLOCK ADDRESS - This is the largest key block address found for each key. The key file end-of-file should never be less than the largest block address for the file plus the number of sectors per key block. *The largest block address for DATAFIL is 18 (the largest block address for DATAFIL is 18 (the largest block address of key 3). Since the number of sectors per block is 8, the key file end-of-file should be at least 26 (see VERIFY output below).*

```
WHICH (1=FILE INFO,   2=KSAM PARAMETERS,   3=KSAM CONTROL?   4=ALL)?3

DATA FILE = DATAFIL       VERSION= A.2.1
KEY CREATED=292/'78   10:19: 7.4      KEY ACCESS= 107/'79 12: 0: 2.9
KEY CHANGED= 93/'79   14:18: 7.6      COUNT START=292/,78 10:19:53.6
DATA RECS  =          20 DATA BLOCKS=       19 END BLK WDS=        19
DATA BLK SZ=          19 DATA REC SZ=       38 ACCESSORS=           0
FOPEN                  2 FREAD               0 FCLOSE               2
FREADDIR               0 FREADC              0 FREADBYKEY           0
FREMOVE                0 FSPACF             57 F'FINDBYKEY          0
FGETINFO               2 FGETKEYINFO         1 FREADLABEL           0
FWRITELABEL            0 FCHECK              0 FFTNDN               3
FWRITE                20 FUPDATF             0 FPOINT               0
FLOCK                  0 FUNLOCK              0 FCONTROL             0
FSETMODE               0
KEYBLK BEAD            7 KFYBLK WROTE        0 KEYBLK SPLTT         0
KEY FILE EOF          26 FREE KEY HD         0 SYSTEM FAILURE       0
                         |-----------------file end-of-file for DATAFIL
```

```
MIN PRIME            11 MAX PRIME           5 RESET DATE      67/'79
DATA FIXED         TRUE DATA B/F            1 TOTAL KEYS           3
FIRST PECNUM          0 MIN PECSIZE        38
```

RECOVERING AFTER SYSTEM FAILURE

KEYINFO only performs the recovery operations if there has been a system failure or if you specify the RECOVER parameter.

If there has been a system failure while the KSAM file is open for non-read access, a flag is set that prevents the file from being opened. Whenever this occurs, KEYINFO must be used in order to reset this flag so that the file can be opened. KEYINFO also recovers from any damage done to the file as a result of the system failure. It resets end-of-file markers for both the data and key files, and deletes any key values that point to records beyond the data file end-of-file. It also stores in the key file the user, group, account, and home group of the user who runs KEYINFO to recover the file. (When there has been a system failure or when KEYINFO is run with the RECOVER option, the KSAM file is opened for exclusive access; otherwise it is opened for shared access.)

When KEYINFO is run after a system failure, the SYSTEM FAILURE count displayed by option 3 of VERIFY is incremented by 1. If there was no system failure but KEYINFO was run with the RECOVER option, this count is not incremented.

When KEYINFO resets the "crash" flag, the date of this reset is saved and can be recovered through the VERIFY command, option 3 under the heading RESET DATE. Note that the NOCHECK option of VERIFY allows that command to open a KSAM file for read-only access even if a system failure prevents the file from being opened for all other access.

For example, assume a file TEST that was open when a system failure occurred. In this case, KEYINFO must be run. Also, assume the following:

- The data file end-of-file (at the end of the data) is beyond the MPE end-of-file (not yet written to file when system failed).
- There are key values beyond the key file end-of-file (internal key file EOF).
- There are data values with no associated key values.

Running KEYINFO will correct the end-of-file markers and, if any keys point to data records beyond the data file end-of-file, it will delete these key values. KEYINFO cannot, however, restore missing key values. To do this, you must reload the file with FCOPY. To illustrate, KEYINFO operates as shown below:

```
>KI TEST
RECOVERY BEGINS

DATA FILE EOF DAMAGED                          reset end-of-file
                                               for data file
DATA FILE MPE EOF HAS BEEN RESET TO KSAM EOF


-------- INFO FOR KEY                   1 ---------

# OF LEVELS OF B-TREE                            2
# OF KEY BLOCKS                                16
# OF SECTORS PER KEY BLOCK                      8
# OF KEYS IN ROOT KEY BLOCK                    14
# OF KEYS IN B-TREE                          1000<------|
# OF KEY BLOCK UTILIZATION                     52.1     |
THE LARGEST KEY BLOCK ADDRESS                 210       |
                                                        |
--------- INFO FOR KEY                   2 ---------    # of keys
                                                        should match
# OF LEVELS OF B-TREE                            2       |
# OF KEY BLOCKS                                 11       |
# OF SECTORS PER KEY BLOCK                       8       |
# OF KEYS IN ROOT KEY BLOCK                      9       |
# OF KEYS IN B-TREE                            997<------|
# OF KEY BLOCK UTILIZATION                      68.6
THE LARGEST KEY BLOCK ADDRESS                  202

WARNING: THERE ARE SOME RECORD(S) WITH KEY VALUE(S) MISSING
          OR KEY VALUE(S) POINTING TO DATA RECORD BEYOND EOF

KEY FILE EOF(INTERNAL) DAMAGED
                                          reset key file end-of-file
KEY FILE (INTERNAL)EOF HAS BFEN RESET

--------- KEY SEQUENCE                    1---------
# OF INVALID KEY VALUES DELETED                 10  keys pointing to non-existent
                                                    data records are deleted
--------- KEY SEQUENCE                   2 ---------
# OF INVALID KEY VALUES DELETED                 10

RECOVERY ENDS


WARNING: THERE ARE SOME RECORD(S) WITH KEY VALUE(S) MISSING
          THE KSAM FILE HAS TO BE RELOADED
```

In this case, the file must be reloaded in order to add the missing key values to the key file. For a full discussion of recovery procedures in case of system failure, including how to reload your file, refer to appendix E.

USING RECOVER OPTION

Even if a system failure does not occur, you can run KEYINFO with the RECOVER option in order to check the file structure.

The RECOVER option forces KEYINFO to correct any end-of-file inconsistency, including the key file end-of-file, and to delete any invalid key values. This option sets the RESET DATE field of the VERIFY output to the current date, and saves your user name, account,

group, and home group, but does not increment the SYSTEM FAILURE count displayed by VERIFY.

Note that checking each record and key in a file with a lot of data is very time consuming. Therefore, you should not use RECOVER unless it is necessary to reconstitute your file.

For example, use KEYINFO with RECOVER to validate file TEST:

```
>KI TEST;RECOVER
RECOVERY BEGINS


--------- INFO FOR KEY                    1 ----------

# OF LEVELS OF B-TREE                              1
# OF KEY BLOCKS                                    1
# OF SECTORS PER KEY BLOCK                         8
# OF KEYS IN ROOT KEY BLOCK                       10
# OF KEYS IN B-TREE                               10
% OF KEY BLOCK UTILIZATION                         4.9
THE LAPGEST KEY BLOCK ADDRESS                      2


--------- KEY SEQUENCE                    1 ----------
# OF INVALID KEY VALUES DELETED                    0

RECOVERY ENDS
```

If you now run VERIFY, using option 3, you will see that the date of recovery is displayed following the heading RESET DATE.

```
>V

WHICH (1=FILE INFO,  2=KSAM PARAMETERS,  3=KSAM CONTROL,  4=ALL)?3

DATA FILE = TEST         VERSION= A.2.4
KEY CREATED= 86/'79  13:55:23.6      KEY ACCESS= 114/'79 14: 1:14.9
KEY CHANGED=114/'79  13:55:48.8      COUNT START= 96/'79 13:55:49.2
DATA RECS  =          10 DATA BLOCKS=          9 END BLK WDS=          8
DATA BLK SZ=           8 DATA REC SZ=         16 ACCESSORS=            0
FOPEN                  2 FREAD                 0 FCLOSE                2
FPEADDIR               0 FPEADC                0 FREADBYKEY            0
FPEMOVE                0 FSPACE                9 FFINDBYKEY            0
FGETINFO               2 FGETKEYINFO           1 FREADLABEL            0
FWRITELABEL            0 FCHECK                0 FFINDN                1
FWRITE                10 FUPDATE               0 FPOINT                0
FLOCK                  0 FUNLOCK               0 FCONTROL              0
FSETMODE               0
KEYBLK READ            3 KEYBLK WROTE          1 KEYBLK SPLIT          0
KEY FILE EOF          10 FREE KEY HD           0 SYSTEM FAILURE        0
MIN PRIME              0 MAX PRIME             9 RESET DATE      114/'79
DATA FIXED          TRUE DATA B/F              1 TOTAL KEYS            1
FIRST RECNUM           0 MIN PECSIZE           2
```

USING KSAMUTIL IN BATCH MODE

A batch job can be developed on the text editor (EDITOR) and then executed with the MPE :STREAM command. In order to distinguish the MPE commands within a streamed batch job from those external to the job, an exclamation point (!) is used as the command prefix rather than a colon (:). KSAMUTIL commands have no command prefix when executed in a batch job.

In the job illustrated in Figure 2-1, the first step after job initialization with the !JOB command is to purge all KSAM and non-KSAM files that will be created within the job. This insures that there are no files in the account with names duplicating files that will be created programmatically with the job.

Following initialization, the first program in the job is run. Since this program uses a KSAM file, MANPN, this file is created with the KSAMUTIL BUILD command before the program is executed. Note that the program is purged immediately before calling BUILD to create it. This is done to make sure that no duplicate key or data file exists in the account.

The newly created KSAM file is used in program MAN1, which was previously compiled and is input from the file CARDIN; output from the program goes to the file REPORT associated with the line printer. An !EOD command follows the program. If data is entered here rather than from the input file, then the !EOD follows the data. Any other programs in the job stream follow the !EOD each with its own terminating !EOD. The entire job is terminated by an !EOJ command.

Figure 2-1 is an EDITOR listing of a job entered to a file through the EDITOR program. This job could also have been punched on cards or any other device that accepts jobs, but in that case, the standard command prefix, the colon (:), would be used. (Refer to the *EDIT/3000 Reference Manual* for instructions on using the EDITOR.)

In order to run this job, you can enter the command:

:STREAM *filename*

where *filename* identifies the EDITOR file where the job was saved.

Batch jobs need not be streamed, but can be entered entirely as a card deck or through some other input device. Streaming allows you to develop and execute the job interactively at your terminal. For a full discussion of using the :STREAM command to introduce jobs in a session, refer to the *MPE Commands Reference Manual*.

Batch jobs do not support the OFFLINE parameter for the KEYDUMP, KEYINFO, KEYSEQ, and VERIFY commands.

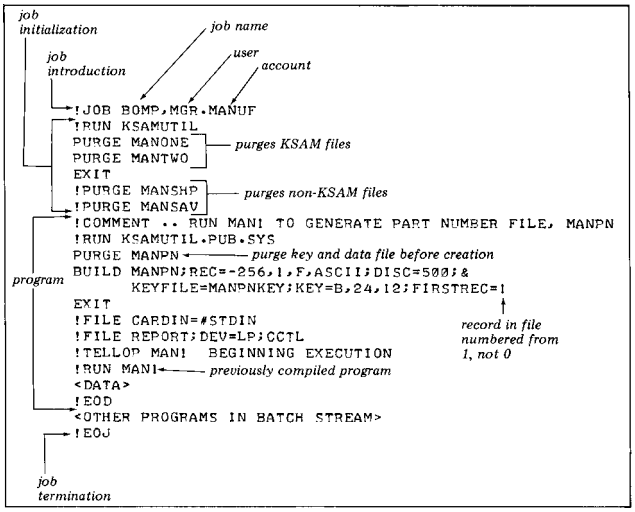


Figure 2-1. EDITOR Listing of Job to be Streamed

FCOPY UTILITY

FCOPY/3000 is the standard HP 3000 utility program that allows you to copy data from one file to another, creating a new KSAM file is desired, to copy selected data, to make multiple copies of the same file, or to display data in a variety of formats. With some exceptions, the same FCOPY functions used to copy other HP 3000 files can be used to copy KSAM files.

Table 2-4 contains a summary of the FCOPY function parameters that apply to copying KSAM files. The function parameters: SKIPEOF,IGNERR, and SUBSET alone, are not included in this list since they are not applicable to KSAM files. Two functions are included that apply only to copying from KSAM files: these are KEY= and NOKSAM. Otherwise, this table includes all the standard FCOPY functions. Note that this summary is meant only to refresh your memory and that it assumes a knowledge of FCOPY. (For a complete description of FCOPY and its operation, refer to the *FCOPY/ 3000 reference manual*.)

RUNNING FCOPY

The FCOPY utility program is executed by the MPE command:

```
:RUN FCOPY.PUB.SYS
```

Program FCOPY prompts for command input with a greater-than sign (>) in column 1 of the next line. You may then enter an FCOPY command in response to the prompt. If you are executing FCOPY in a batch job rather than in a session, you enter the command (omitting the prompt) in column 1 of the line following the :RUN FCOPY.PUB.SYS command.

EXITING FROM FCOPY

In order to terminate FCOPY and retum to the MPE Operating System, enter the following command:

```
>EXIT or >E
```

FCOPY FROM COMMAND

Copies data from one file to another.

```
> FROM= [fromfile] ;TO= [tofile] [;functionlist]
```

The FROM command specifies the file from which data is copied and the file to which it is copied. It optionally includes one or more function specifications in the *functionlist* parameter.

PARAMETERS

fromfile Specifies the file to be copied. For a KSAM file this should be the actual file designator. An asterisk (*), indicating the "from" file designated in the immediately preceding FCOPY command should *not* be used to copy KSAM files. If *fromfile* is omitted, the standard input device $STDIN is assumed.

tofile Specifies the file to receive the data. For a KSAM file this should be the actual file designator. If *tofile* is specified as (*dfile,kfile*) where *dfile* is a data file name and *kfile* is a key file name, a new KSAM file is created with the same characteristics as the *fromfile*. The data and key values are copied from the existing file to the new file excluding any data records tagged for deletion.

If *tofile* is omitted, the standard list device $STDLIST is assumed. Using this device as a "tofile" is a good way to display the contents of a KSAM file at your terminal during a session and on the line printer in a batch job.

functionlist One or more keyword parameters separated by semicolons that specify particular FCOPY functions. (Refer to Table 2-4 for a complete list.)

KSAM OPTIONS

Two keyboard options may be used with FCOPY to copy KSAM files: the KEY= option and the NOKSAM option. These two options are mutually exclusive; they cannot both be specified in the same FCOPY FROM command. When neither option is specified, the KSAM *fromfile* is copied to another file in primary key sequence. This is exactly like copying any HP 3000 file to another with FCOPY.

Table 2-5 summarizes the results of using, or omitting, the KSAM options KEY= and NOKSAM.

KEY= OPTION. KEY= specifies a key whose value determines the sequence in which the file is copied. The object of KEY= is a positive integer that identifies the key by its starting character location in the data file. The indicated key may be either the primary or an alternate key. If the object of KEY= is zero, then the file is copied in chronological sequence rather than in key sequence.

If KEY= and NOKSAM are both omitted, the KSAM file is copied in primary key sequence. In this case and in the case where KEY= is specified, only active records, not those tagged for deletion are copied.

Table 2-4. FCOPY Functions with KSAM Files

FUNCTION LIST ENTRY*	ACTION PERFORMED
;KEY=*nn*	Copy active records from KSAM file in sequence by key located at *nn*; if omitted, copy file in primary key sequence; if *nn* is zero, copy file in chronological order. The KSAM EOF is used.
;NOKSAM	Copy all records, including deleted records, from data file of KSAM file to any other file. Copy is in chronological sequence; records must be fixed length. The data file being copied is opened as an MPE file and the MPE end-of-file is used. Unless the TO file is an MPE file created with l user label, specify NOUSER LABELS.
;NEW	Copy active records and associated key values from KSAM file to new KSAM file specified as TO= (*dfile,kfile*).
{ ;EBCDICIN / ;BCDICIN } { ;EBCDICOUT / ;BCDICOUT }	Translate copied data from EBCDIC to BCDIC code to ASCII. Translate copied data from ASCII code to EBCDIC or BCDIC.
;UPSHIFT ;SUBSET={ *"string"* / *#pattern#* } [, [*column*] / [,EXCLUDE]] ;SUBSET=[*first-record*] [{ ,*#records* / :*last record* }]	Convert any copied lower-case characters to uppercase. Copy from data file only records containing specified *"string"* or *#pattern#* starting search in specified *column* or column 1. If EXCLUDE specified, copy all data file records except those containing *"string"* or *#pattern#* Copy from data file as many records as are specified in *#records* starting with *first-record*; or copy from *first-record* through *last-record* inclusive. If *first- record* omitted, start at first sequential record in file; if *#records* or *last-record* omitted, copy through last sequential record in file.
;VERIFY =*#errors*] ;COMPARE [= *#errors*]	Verify accuracy of copy where both files are on disc; terminate if *#errors* exceeded; Compare without copying the *fromfile* to the *tofile*, terminate if differences exceed *#errors*. Comparison applies only to KSAM data files.

Table 2-4. FCOPY Functions with KSAM Files (continued)

FUNCTION LIST ENTRY*	ACTION PERFORMED
;OCTAL [;CHAR] [;NORECNUM] [;TITLE="*title*"] ;HEX [;CHAR] [;NORECNUM] [;TITLE="*title*"] ;CHAR [{;OCTAL / ;HEX}] [;NORECNUM] [;TITLE="*title*"]	Display contents of "from" file as octal images on a word-by-word basis. Display contents of "from" file as hexadecimal images on a word-by-word basis. Display contents of "from" file as character images on a word-by-word basis.

*IGNERR, SKIPEOF, and SUBSET without parameters do not apply to "from" files.

Table 2-5. KSAM Options of FCOPY

OPTION	RESULT
KEY=omitted	Copy KSAM file in primary key sequence.
KEY=*n* (*n*>0)	Copy KSAM file in sequence by the key (primary or alternate) located starting at character *n* of each data record (counting from first character = 1).
KEY=0	Copy KSAM file in chronological sequence (the sequence in which records were actually stored in the file); copy *excludes* records marked for deletion.
NOKSAM	Copy the data file of a KSAM file to any file in chronological sequence; copy *includes* records marked for deletion.

NOKSAM OPTION. NOKSAM allows you to copy the data file of a KSAM file with fixed-length records to any MPE file, including KSAM files. All indicated records of the data file are copied, including those tagged for deletion. When you copy a file using the NOKSAM option, you should also specify the NOUSERLABELS option. The only exception to this rule is if the TO file is an MPE file that you have already created with one user label.

USING FCOPY

FCOPY is useful in order to compact a KSAM file that has many records tagged for deletion. When a file has been used for a period of time, changes and deletions may result in a high percentage of inactive records. In order to recover the space occupied by such records, you can copy the file to a new file with FCOPY. Since FCOPY copies only active records, records that are not tagged for deletion, the new KSAM file has no unused space embedded among the data records.

FCOPY can also be used to recover records tagged for deletion in a KSAM file. The FCOPY NOKSAM option copies *all* records including those tagged for deletion. The first two characters of such records will contain the delete code rather than their original values, but otherwise are recovered intact. This can be a useful feature in order to recover records deleted by mistake.

Another use of FCOPY is to reload data from a damaged file to a new file. This may be required as a result of a system failure. If you decide to reload a KSAM file following a system

failure, you should first run the KEYINFO command of KSAMUTIL to reset the end-of-file markers and delete any invalid key values. If the file is still damaged and you choose to reload it, you should use FCOPY to transfer existing records to a new undamaged KSAM file. In this case, you use the KEY=0 option rather than the NOKSAM option, unless you want to keep all the deleted records or the key file was lost.

FCOPY WITH NO OPTIONS. Assume a file named KSAMFILE created with one primary key, an integer located at character 21. Since many records were tagged for deletion in the file, it is time to copy the active records to a new file, You may either create a new KSAM file with the BUILD command as shown in example 1, or use FCOPY to create the new KSAM file as shown in example 2. In either case, you should purge the original file (KSAMFILE in the examples) and then rename the new file (KSAMFIL2) with the data and key file names of the original file so that any programmatic references to the file need not be changed.

You may also use FCOPY to create an empty KSAM file with all the characteristics of an existing file, but with no data. The method for doing this is shown in example 3.

1. Create new file with BUILD:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD KSAMFIL2;KEYENTITY =KFIL2;KEY=I,21,2 <-------- create "to"file
>EXIT
:RUN FCOPY.PUB.SYS
>FROM=KSAMFILE;TO=KSAMFIL2 <----------- copy in primary key sequence
>EXIT
:RUN KSAMUTIL.PUB.SYS
>PURGE KSAMFILE <----------------- purge "from" file after copy
KSAMFILE.KSAM.DATAMGT & KFILE PURGED
>RENAME KSAMFIL2,KSAMFILE|
                         |  <--------- rename copied file with old file names
>RENAME KFIL2,KFILE      |
>EXIT
:
```

2. Use FCOPY to create new file:

```
:RUN FCOPY.PUB.SYS
>FROM=KSAMFILE;TO=(KSAMFIL2,KFIILE2)
>EXIT
:RUN KSAMUTIL.PUB.SYS
>PURGE KSAMFILE
KSAMFILE.KSAM,DATAMGT & KFILE PURGED
>RENAME KSAMFIL2,KSAMFILE
>RENAME KFILE2,KFILE
>EXIT
:
```

   You may specify the ;NEW function in the FCOPY FROM command for purposes of documentation. Its inclusion or omission does not affect the command in any way.

   This method not only creates the new KSAM file, but also copies all the data from the existing file to the new file (except records marked for deletion). Example 3 below, shows how you can create a KSAM file with exactly the same specifications as an existing file but with no data.

3. Use FCOPY to build a new file with no data:

```
:RUN FCOPY.PUB.SYS
>FROM=KSAMFILE; TO=(KSAMFIL3,KFILE3); SUBSET=1,0 <---- copy 0 records
0 RECORDS PROCESSED *** 0 ERRORS
```

The new file, KSAMFIL3, is created with exactly the same specifications as the existing file KSAMFILE, but with no data. This is easier than building the file with the BUILD command, but should be used only if the new file is to have keys in the same position and the same length as the existing file.

Following any of these operations, only active records are contained in the new KSAM files. These records are stored in primary key sequence in the data file; that is, the new chronological and the primary key sequences axe the same. If you prefer to maintain the original chronological sequence, then you can use the KEY=0 option.

4. Use FCOPY to add data to an existing file:

   Before running FCOPY to add new records to a file that contains data, make sure that file (the TO file) is opened for either APPEND or INOUT access. Otherwise, FCOPY will open the TO file for write-only access causing the end-of-file to be reset to zero and any existing data to be lost. For example:

```
:FILE A = KSAMFILE,OLD;ACC = APPEND
                              or
                          ACC = INOUT
:RUN FCOPY.PUB.SYS
>FROM = NEWDATA ;TO = *A
```

   The data in the file NEWDATA is appended to the data in the existing file, KSAMFILE, in primary key sequence (the default).

FCOPY WITH KEY = OPTIONS.

1. Assume that a company's employee records have been maintained in sequence by social-security-number in a KSAM file, EMPLOY, but a new policy requires that they be maintained in sequence by employee number. FCOPY can be used to transfer the data to a new file, EMPLOY2, in which all employees are re-ordered by their unique employee numbers.

   Assume EMPLOY was created with the following command:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD EMPLOY;REC=3000;KEYENTITY =EMPKEY;&
>      KEY=B,3,11;& <--------------- primary key (social-security-number)
>      KEY=B,14,5;& <------------ alternate key (employee number)
>      KEY=B,19,30,,DUP <------ alternate key (name)
>EXIT
```

   Before copying and resequencing file EMPLOY, a new KSAM file is built:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD EMPLOY2;REC=3000;KEYENTITY =EMPKEY2;&
>      KEY=B,14,5;& <------------ primary key (employee number)
>      KEY=B,19,30,,DUP <----- alternate key (name)
>EXIT
```

   There is no need for the new key file to retain the same structure as the key file of the copied file. The primary key in EMPLOY has been dropped from EMPLOY2; although the socialsecurity-number remains in the data file, it is no longer a key. An alternate key in EMPLOY, the employee's identification number, is the primary key in EMPLOY2.

   Once the new KSAM file has been created, you can copy the old file EMPLOY to the new file EMPLOY2 in the new sequence:

```
:RUN FCOPY.PUB.SYS
>FROM=EMPLOY;TO=EMPLOY2;KEY=14
>EXIT                        |
     |----------------------|
     |column number of key used to sequence EMPLOY2
```

   To avoid changing programs that reference the file EMPLOY, you can rename EMPLOY2 with the name EMPLOY, first purging the old file EMPLOY:

```
:RUN KSAMUTIL.PUB.SYS
>PURGE EMPLOY
KSAMFILE EMPLOY.KSAM.DATAMGT & EMPKEY PURGED
>RENAME EMPLOY2,EMPLOY <--------------- rename data file
>RENAME EMPKEY2,EMPKEY <--------------- rename key file
>EXIT
```

2. Another use of FCOPY is to copy a selected portion of one KSAM file to another. For example, using the same file EMPLOY used in previous examples, you can copy all the employee records whose last names begin with the letter A into a new file sequenced by employee name:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD EMPLOYA;KEYENTITY =AKEY;KEY=B,19,30,,DUP
>EXIT
:RUN FCOPY.PUB.SYS
>FROM=EMPLOY;TO=EMPLOYA;KEY=19;SUBSET="A",19
>EXIT
```

The new file EMPLOY is sequenced by the key starting in column 19 (employee name) and only contains records for employees whose last names start with A.

3. If you want to copy the KSAM file in chronological sequence, you can use the KEY=0 option. Since this option copies only active records, it can be used to compact a file in which many records are tagged for deletion while retaining the chronological order in which the file was created. It is also the preferred option for reloading a KSAM file after a system failure.

   Assume the new file EMPLOYX has the identical structure to the file EMPLOY used in the previous examples:

```
            :RUN FCOPY.PUB.SYS
>FROM=EMPLOY;TO=EMPLOYX;KEY=0
>EXIT
```

   The new file is identical in its chronological sequence to the old file, but contains only active records.

4. To find out how many records are currently active in a KSAM file, you can use FCOPY as follows:

```
    :RUN FCOPY.PUB.SYS
> FROM=KSAMFILE ;TO=$NULL
 N RECORDS PROCESSED *** 0 ERRORS (where N is the number of active  records
 in the KSAM file)
>EXIT
```

   Only the active records (those not marked for deletion) will be listed as present in the file. (You can also calculate the number of active records by looking at the VERIFY listing, option 3, and subtracting the number of FREMOVEs from the FWRITES.)

FCOPY WITH NOKSAM OPTION.

1. Using NOKSAM, you can copy the data file of a KSAM file to another file. The records are copied in chronological sequence. Since NOKSAM copies records marked for deletion as well as active records, it provides a method for recovering the data in any records marked for deletion. For example, if certain records in file EMPLOY were incorrectly marked for deletion, the NOKSAM option could be used to copy the entire data file to a new file including the inactive records.

Using the SUBSET parameter of FCOPY, you can copy only those records marked for deletion. In the following example, all deleted records are listed on the line printer:

```
                    pattern of all 1's in 1st 2 characters
                                |
                                |
:FILE X; DEV=LP                 |
:RUN FCODY.PUB.SYS      ESC/---------/>FROM=EMPLOY;TO=*X;SUBSET=#%377,%377#,1;NOKSAM;OCTAL;CHAR;NORECNUM;&
>TITLE="RECOPDS DELETED FROM THE FILE, EMPLOY"
```

   When records are deleted from a KSAM data file, a pattern of all 1's is written to the first two characters of the deleted record. (ln each character this pattern can be represented as the octal value %377.) If you want to be able to recover key data from deleted records in this manner, you should avoid placing key data in the first two characters of a data record.

Note that you should not use the NOKSAM option to copy a KSAM file with variable-length records to another KSAM file. Also, if NOKSAM must be used to reload a file after a system crash (for instance, because the key file was lost), you should use the SUBSET option to copy

only valid records. Normally, you use the KEY=0 option to reload KSAM files after a system failure.

(Refer to appendix E, Recovery From System Failure, for a full discussion of using FCOPY to reload a KSAM file following a system failure.)

COPYING VARIABLE-LENGTH KSAM FILES

You can copy a variable-length KSAM data file by first copying it to an MPE file and then copying the MPE file to a new KSAM file.

First use the MPE :LISTF,2 command to list the attributes of the KSAM data file. Then use the MPE :BUILD command to build an MPE file with a record length two words less than that of the KSAM data file, a blocking factor the same as the data file (always 1), variable-length records, and the same record limit as the KSAM data file.

Copy the KSAM data file to the MPE variable-length file on a block-by-block basis, as follows:

```
      :FILE DATAFILE; NOBUF
:FILE MPEFILE; NOBUF
:FCOPY FROM= *DATAFILE; TO= *MPEFILE; NOKSAM;NOUSERLABELS
```

Copy the variable-length records from the MPE file to a newly created KSAM file omitting the deleted records.

```
FCOPY FROM= MPEFILE;TO= KSAMFILE;SUBSET= #%377, %377#,,EXCLUDE
```

DISPLAY COPIED FIIES ON $STDLIST. When you omit the "to" file from the TO= specification, the standard output device is assumed. This allows you to list the contents of the KSAM file at your terminal in a session or on the line printer in a job.

Assume the file JNAMES with a primary key (last name) starting in character 11 and three alternate keys: a phone number starting in character 21, a city name starting in character 53, and a zip code starting in character 67.

1. *If KEY= is omitted, the file is listed in primary key, order:*

```
>FROM=JNAMES:TO=;KEY=)
JEANNE   /ALOGICAL   226-0295 4942  COUSIN CT         SUNNYVALE    95054
POLLY    |CHROMATIC  267-1412 1148  COLORFUL CT       SAN JOSE     95130
ANNA     |FORA       253-5246 9283  TROCHAIC TRAIL    SAN JOSE     95131
ANNE     |HOWE       372-4328 6547  EXUBERANCE WY     CAMPBELL     95112
HY       |KUVERSE    267-8961 650   LOTUS BLOSSOM WY  SAN JOSE     95136
ANNA  |--|LOGUE      224-8934 1707  INVERSE WY        MOUNTAINVIEW 95051
ARTHUR|  |MOMITER    443-5346 1554  MERCURY ST.       MILPITAS     94173
CLARA |  |NETTE      243-4493 2667  GOODMAN DR        ALVISO       95143
RHEA  |  |PREYSELLE  365-8551 10879 REVIEW ROAD       SAN JOSE     95070
KURT  |  |REMARQUE   243-1043 34    BRIEF ST          MILPITAS     94062
MIKE  |  |ROMETER    269-1712 1681  MACHINIST         SUNNYVALE    95112
TRUDY |  \TEKTIFF    255-1005 17155 POIROT PL         CAMPBELL     95121
EOF FOUND IN FROMFILE AFTER RECORD 11
      |
      |
12 RECORDS PROCESSED *** 0 ERRORS
      |
      |
      |
      ascending order by primary key
```

   Use SUBSET to list selected portions of the file, for example, to list the first two records in primary key sequence:

```
>FROM=JNAMES;TO=;KEY=11;SUBSET=0,2
JEANNE    ALOGICAL  226-0295 4942  COUSIN CT    SUNNYVALE   95054
POLLY     CHROMATIC 267-1413 1148  COLORFUL CT  SAN JOSE    95030

          2 RECORDS PROCESSED *** 0 ERRORS
```

2. If KEY= *"primary key location", the file is listed in primary hey order:*

```
>FROM=JNAMES:TO=;KEY=)
JEANNE    ALOGICAL   226-0295 4942  COUSIN CT         SUNNYVALE    95054
POLLY     CHROMATIC  267-1412 1148  COLORFUL CT       SAN JOSE     95130
ANNA      FORA       253-5246 9283  TROCHAIC TRAIL    SAN JOSE     95131
ANNE      HOWE       372-4328 6547  EXUBERANCE WY     CAMPBELL     95112
HY        KUVERSE    267-8961 650   LOTUS BLOSSOM WY  SAN JOSE     95136
ANNA      LOGUE      224-8934 1707  INVERSE WY        MOUNTAINVIEW 95051
ARTHUR    MOMITER    443-5346 1554  MERCURY ST.       MILPITAS     94173
CLARA     NETTE      243-4493 2667  GOODMAN DR        ALVISO       95143
RHEA      PREYSELLE  365-8551 10879 REVIEW ROAD       SAN JOSE     95070
KURT      REMARQUE   243-1043 34    BRIEF ST          MILPITAS     94062
MIKE      ROMETER    269-1712 1681  MACHINIST         SUNNYVALE    95112
TRUDY   |>TEKTIFF    255-1005 17155 POIROT PL         CAMPBELL     95121
EOF FOUND IN FROMFILE AFTER RECORD 11
        |
        |
12 RECORDS PROCESSED *** 0 ERRORS
        |
        |
        |
        | byte location 11 (same sequence as previous example)
```

3. *If KEY= "alternate key location", the file is listed in sequence by that key:*

```
>FROM=JNAMES; TO=; KEY=67 /
KURT    REMARQUE   243-1043 34    BRIEF ST          MILPITAS      /94062
ARTHUR  MOMITER    443-5346 1554  MERCURY ST        MILPITAS      |94173
DOLLY   CHROMATIC  267-1413 1148  COLORFUL CT       SAN JOSE      |95030
ANNA    LOGUE      224-9934 1707  INVERSE WY        MOUNTAIN VIEW |95051
JEANNE  ALOGICAL   226-0295 4942  COUSIN CT         SUNNYVALE     |95054
RHEA    PREYSELLE  365-9551 10879 REVIEW ROAD       SAN JOSE      |95070
ANNE    HOWE       372-4328 6547  EXUBERANCE WY     CAMPBELL      |95112
MIKE    ROMETER    269-1712 1681  MACHINIST DR      SUNNYVALE     |95112
TRUDY   TEKTIFF    255-1005 17155 POIROT PL         CAMPBELL  |---|95121
ANNA    FORA       253-5246 9283  TROCHAIC TRAIL    SAN JOSE  |   |95131
HY      KUVERSE    267-8961 650   LOTUS BLOSSOM WY  SAN JOSE  |   |95136
CLARA   NETTE      243-4493 2667  GOODMAN DR        ALVISO    |   \95143
EOF FOUND IN FROMFILE AFTER RECORD                             |    |
                                                               |    |
12 RECORDS PROCESSED *** 0 ERRORS                              |    |
                                                               |    |
                              output in ascending order--------|    |
                            by key in byte location 67-------------|
```

Use SUBSET= to list all the records with the characters "SUNNYVALE" starting in column 53; sequence is by alternate key in location 67:

```
>FROM=JNAMES;TO=;KEY=67;SUBSET="SUNNYVALE",53

JEANNE    ALOGICAL  226-0295 4942  COUSIN CT     SUNNYVALE   95054
MIKE      ROMETER   269-1712 1681  MACHINIST DR  SUNNYVALE   95112
EOF FOUND IN FROMFILE AFTER RECORD 11

 2 RECORDS PROCESSED *** 0 ERRORS
```

Another example using SUBSET= lists five records starting with the fourth record; sequence is by alternate key in location 67:

records numbered from O

```
>FROM=JNAMES;TO=;KEY=67;SUBSET=3,5
ANNA    LOGUE       224-9934 1707   INVERSE WY     MOUNTAIN VIEW  95051
JEANNE  ALOGICAL    226-0295 4942   COUSIN CT      SUNNYVALE      95054
RHEA    PREYSELLE   365-8551 10789  REVIEW ROAD    SAN JOSE       95070
ANNE    HOWE        372-4328 6547   EXUBERANCE WY  CAMPBELL       95112
MIKE    ROMETER     269-1712 1681   MACHINIST DR   SUNNYVALE       95112
5 RECORDS PROCESSED *** 0 ERRORS
```

4. *If KEY=O, the file is copied in chronological order:*

```
>FROM=JNAMES; TO=; KEY=0
ARTHUR   MOMITER   443-5346 1554   MERCURY ST        MILPITAS      94173
TRUDY   TEKTIFF    255-1005 17155  POIROT PL         CAMPBELL      95121
ANNA    LOGUE      224-8934 1707   INVERSE WY        MOUNTAIN VIEW 95051
CLARA   NETTE      243-4493 2667   GOODMAN DR        ALVISO        95143
ANNE    HOWE       372-4328 6547   EXUBERANCE WY     CAMPBELL      95112
JEANNE  ALOGICAL   226-0295 4942   COUSIN CT         SUNNYVALE     95054
HY      KUVERSE    267-8961 650    LOTUS BLOSSOM WY  SAN JOSE      95136
MIKE    ROMETER    269-1712 1681   MACHINIST DR      SUNNYVALE     95112
ANNA    FORA       253-5246 9283   TROCHAIC TRAIL    SAN JOSE      95131
POLLY   CHROMATIC  267-1413 1148   COLORFUL CT       SAN JOSE      95130
RHEA    PREYSELLE  365-9551 10879  REVIEW ROAD       SAN JOSE      95070
KURT    REMARQUE   243-1043 34     BRIEF ST          MILPITAS      94262
EOF FOUND IN FROMFILE AFTER RECORD 11

        12 RECORDS PROCESSED *** 0 ERROR
```

:STORE AND :RESTORE COMMANDS

The :STORE and :RESTORE commands are used primarily to provide back-up for user disc files. The file or set of files is copied to magnetic tape or serial disc by the :STORE command in a special format that permits the serial device to be read back onto disc with the :RESTORE command. The use of these two commands for KSAM files is identical to their use with any HP 3000 files. (Refer to the *MPE Commands Reference Manual* for a complete description of the :STORE and :RESTORE commands.)

STORE

Stores KSAM file on magnetic tape or serial disc.

:STORE *filereference1*,*filereference2* [,...] ; *storefile*

[;SHOW] [;FILES=*maxfiles*]

This command is used to store one or more disc files onto magnetic tape or serial disc. *When used to store KSAM files, both the data file and the key file must be specified.*

PARAMETERS

filereference1 Actual file designator of data file; specified in the following format:

filename [/*lockword*] [.*groupname* [.*accountname*]]

where each subparameter is a name consisting of from 1 to 8 alphanumeric characters beginning with a letter.

(Required parameter for KSAM files.)

filereference2 Actual file designator of key file; specified in exactly the same format as *filereference1*.

(Required parameter for KSAM files.)

storefile Name of destination device file onto which the stored files are written. This can be any magnetic tape or serial disc file from the output set. This file must be referenced in the back-reference (*) format; this format references a previous :FILE command that identifies the file as a magnetic tape or serial disc file.

(Required parameter.)

SHOW Request to list names of file stored. If SHOW is omitted, total number of files stored, names of files not stored, and number of files not stored are listed.

(Optional parameter.)

FILES=*maxfiles* Maximum number of files that may be stored. If omitted, 4000 is specified by default.

(Optional parameter.)

USING THE :STORE COMMAND

Before issuing a :STORE command, you must identify the *storefile* as a magnetic tape or as a serial disc with the :FILE command using the following format:

:FILE *formaldesignator* [=*filereference*] ;DEV=*device*

The *device* parameter must indicate the device class name or logical unit number of a magnetic tape or serial disc unit, All other parameters for *storefile* are supplied by the :STORE command executor; if you attempt to supply any of these yourself, MPE rejects the :STORE command.

If you press the BREAK key during the store operation, the operation stops after storing the current file and further output is suppressed.

For example, to copy KSAM file KSAMDATA to a magnetic tape file named SAVE FILE

```
:FILE T=SAVEFILE;DEV=TAPE
:STORE KSAMDATA,KSAMKEY;*T
        ^         ^
        |         |
        |         |
   data file   key file
```

Note that *both* the data and key file must be specified in order to store the entire KSAM file.

If you want to copy this same file to a serial disc, use the following command sequence:

```
:FILE SD=SAVEFILE;DEV=SDISC
:STORE KSAMDATA, KSAMKEY;*SD
```

RESTORE

Restores KSAM file from magnetic tape or serial disc.

```
:RESTORE restorefile ;filereference1,filereference2[, . . .] [;KEEP]
      [;DEV=device] [;SHOW] [;FILES=maxfiles]
```

Restores to disc, one or more files stored off-line to magnetic tape or serial disc by the :STORE command. *To restore a KSAM file, both the data file and the key file names must be specified.*

PARAMETERS

restorefile Name of magnetic tape or serial disc file on which files to be retrieved now reside. This file must be referenced in the backence (*) format; this format references a previous :FILE command that defines the file as a magnetic tape or serial disc file. A message is output to the Console Operator requesting him to mount the magnetic tape or serial disc platter identified by the *filereference* parameter in the :FILE command, and allocate the tape unit or disc platter to you.

(Required parameter.)

filereference1 Acual file designator identifying the KSAM data file, specified in the format:

filename [/*lockword*] [.*groupname* [.*accountname*]]

where each subparameter is a name consisting of from 1 to 8 alphanumeric characters beginning with a letter.

(Required for KSAM files.)

filereference2 Actual file designator identifying the KSAM key file, specified in the same format as *filereference1* .

(Required for KSAM files.)

KEEP Specification that if a file referenced in the :RESTORE command currently exists on disc, the file on disc is kept and the corresponding file on tape or serial disc is not copied into the system. If KEEP is omitted, and an identically-named file exists in the system, that file is replaced with the one on the tape or serial disc. If KEEP is omitted, *and* a file on tape or serial disc is eligible for restoring *and* a file of the same name exists on disc, *and* this disc file is busy, the disc file is kept and the tape or serial disc file is not restored.

(Optional parameter.)

DEV=*device* Device class name or logical number of device on which files are to be restored. (This name is also written on the label of each file restored.) If you omit this parameter, MPE attempts to replace the files on a device of the same class (or logical device number) as that of the device on which the file was created, If this attempt fails, perhaps because the device class specified does not exist or the tape or serial disc was created on a previous version of this computer, MPE attempts to replace each file on a disc of the same type (fixed or moving -head) and subtype as that on which it was created. If this fails, MPE attempts to restore the file to a device of class name DISC. If this fails, the file is not restored. If the KSAM file was created with the data file

and the key file on different devices, then RESTORE twice using different DEV=device in each RESTORE.

(Optional parameter.)

SHOW — Request to list names of restored files. If you omit SHOW, only total number of files restored, list of files not restored (and the reason each was not restored), and count of files not restored, are listed.

(Optional parameter.)

FILES=*maxfiles* — Maximum number of files that may be restored. If omitted, 4000 is assigned by default.

(Optional parameter.)

USING THE :RESTORE COMMAND

Before issuing a :RESTORE command you must identify *tapefile* as a magnetic tape or serial disc file with the :FILE command:

```
:FILE formaldesignator [=filereference] ;DEV=device
```

The device parameter must indicate the device class name or logical unit number of a magnetic tape or a serial disc unit. No other parameters than these may be supplied. If you attempt to supply more, the :RESTORE command is rejected.

To retrieve from the magnetic tape file SAVEFILE, the KSAM file KSAMDATA that includes data file (KSAMDATA) and key file (KSAMKEY):

```
:FILE T=SAVEFILE;DEV=TAPE
:RESTORE *T;KSAMDATA,KSAMKEY;KEEP;DEV=DISC;SHOW
```

To retrieve this same file from the serial disc STORDISC, enter the commands:

```
:FILE SD=STORDISC; DEV=SDISC
:RESTORE*SD;KSAMDATA,KSAMKEY;KEEP;DEV=DISC;SHOW
```

Note that both the data file and the key file must be specified in order to restore the entire KSAMI file.

If the KSAM file currently saved on magnetic tape or serial disc was originally created with the data file resident on one device and the key file resident on a different device, then this capability can be retained only if you RESTORE twice using different DEV= specifications in each command.

For example:

```
:FILE T:DEV=TAPE
:RESTORE *T;KSAMDATA;DEV=DISCONE
:RESTORE *T;KSAMKEY;DEV=DISCTWO
```

Upon successful completion, KSAMDATA will be restored from tape file T to a device class identified as DISCONE, and KSAMKEY will be restored from tape file T to a device class identified as DISCTWO. You would do this only in the case where the file was originaly created using the BUILD command specification DEV=DISCONE for the data file, and KEYDEV=DISCTWO for the key file.

3

USING KSAM FILES IN COBOL PROGRAMS

OVERVIEW

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. (Refer to Table 3-1 for a list of the procedures and their associated functions.) The COBOL procedures provided with KSAM/3000 correspond to the INDEXED I-O module statements in COBOL 74.

Note: The following applies when using KSAM with COBOL.

- The KSAM file must be created with KSAMUTIL's >BUILD command.
- To access a KSAM file in chronological order, the KSAM file must be copied to a non-KSAM file.
- KSAM permits duplicate primary keys as an extension to the ANSII standards.

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 start on word boundaries.

Table 3-1. KSAM Procedures for COBOL Interface

PROCEDURE NAME	PARAMETERS	FUNCTION	PAGE
CKCLOSE	*filetable* *status*	Terminates processing of KSAM file identified by *filetable*.	3-12
CKDELETE	*filetable* *status*	Logically removes record from KSAM file; deleted record is identified by previous read.	3-13
CKERROR	*status*, *result*	Converts numeric value returned in *status* to character string *result*.	3-17
CKLOCK	*filetable* *status* *lockcond*	Dynamically locks file opened for shared access, conditionally depending on *lockcond*.	3-18
CKOPEN	*filetable* *status*	Initiates processing of file named in *filetable*; returns file number to first word of *filetable*.	3-20
CKOPENSHR	*filetable* *status*	Initiates processing with dynamic locking and shared access of file named in *filetable*.	3-25
CKREAD	*filetable* *status* *record* *recordsize*	Reads next sequential record from KSAM file identified by *filetable* into *record*.	3-26
CKREADBYKEY	*filetable* *status* *record* *key* *keyloc* *recordsize*	Reads into *record* first record with a key in location *keyloc* whose value matches that of *key*, from KSAM file identified by *filetable*.	3-29
CKREWRITE	*filetable* *status* *record* *recordsize*	Replaces last sequential record read by CKREAD, or replaces record whose primary key matches the value of key item in *record*, with the contents of *record*.	3-32
CKSTART	*filetable* *status* *relop* *key* *keyloc* *keylength*	Positions record pointer in preparation for a sequential read to the first record with a key in location *keyloc* whose value has the relation *relop* to the value of *key*.	3-36
CKUNLOCK	*filetable* *status*	Unlocks file dynamically locked by CKLOCK.	3-40
CKWRITE	*filetable* *status* *record* *recordsize*	Writes record of length *recordsize* from *record* to a KSAM file identified by *filetable*.	3-42

CALLING A KSAM PROCEDURE

The KSAM interface procedures (refer to Table 3-1 for a complete list) are called using a CALL statement of of the following general form:

```
CALL "name" USING filetable,status [,parameter[,. . .] ]
```

Where:

"name"	identifies the procedure to which control is transferred.
filetable	an 8-word 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 word 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 word boundary. In order to assure 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 chronological order or to access a record by its chronological record number. (Chronological 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 words as illustrated in Figure 3-1.

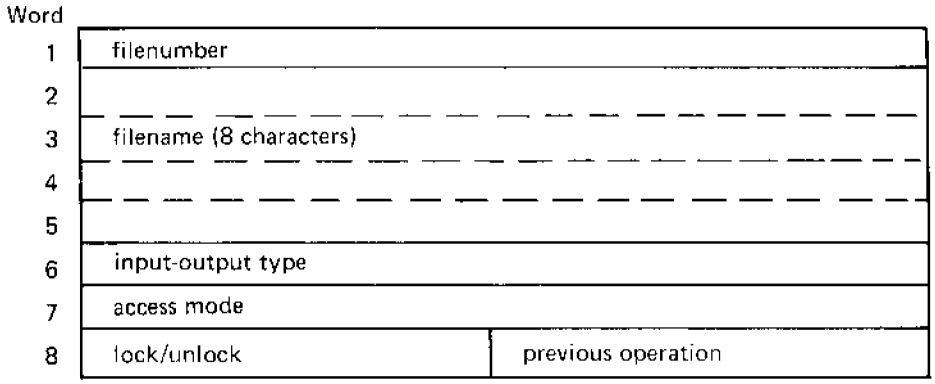


Figure 3-1. Filetable Structure

filenumber — A number identifying the file returned by the CKOPEN procedure after the file named in words 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 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 word 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

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 word 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

EXAMPLE

A sample file table definition might be:

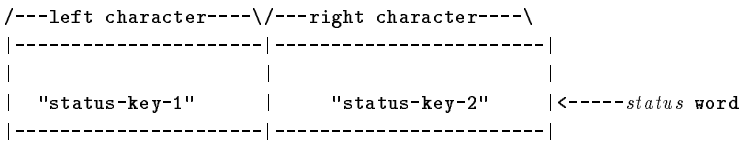
```
WORKING-STORAGE SECTION.

    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.



The possible combinations of the left and right characters of parameter *status* are shown in Table 3-2 The values of status-key-2 (the right character) shown in the table are the only valid values for status-key-2.

Table 3-2. Valid *status* Parameter Character Combinations

If left character of *status* (status-key-1) equals:	**Then right character of *status* (status-key-2) may equal:**
"0" (successful completion)	"0" (no further information) "2" (duplicate key)
"1" (at end)	"0" (no further information)
"2" (invalid key)	"1" (sequence error) "2" (duplicate key) "3" (no record found) "4" (boundary violation)
"3" (request denied)	"0" (lock denied) "1" (unlock denied)
"9" (file system error)	"*n*" where *n* is the MPE file system error code.

Combining status-key-1 with status-key-2, the following values may be returned to the *status* parameter as a whole:

If status = "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.

STATUS

If status = "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 (refer to appendix A). To convert this binary value to numeric display format, call the CKERROR routine (described next in this section).

STATUS

USING STATUS

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 key file that identifies the current record in the data file. 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*. Pointerent 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. (Refer to Table 3-3 for a summary of those procedures that either position the pointer or are dependent on the pointer position.)

Table 3-3. 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)	Pointer remains positioned to key value for record jus modified; *unless* any key value in record was changed, in which case, it points to next key in ascending seuence after the key in the modified record.
	NO (random or dynamic mode)	

SHARED ACCESS

Particular care must be taken when using the logical record pointer during shared access (the file was opened with CKOPENSHR). Since the record pointer is maintained in a separate control block for each open file, if more than one user opens the same file, one user may modify the key file causing the record pointers of other users to point to the wrong key.

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 3-2 is used in all subsequent examples associated with the COBOL procedure calls.

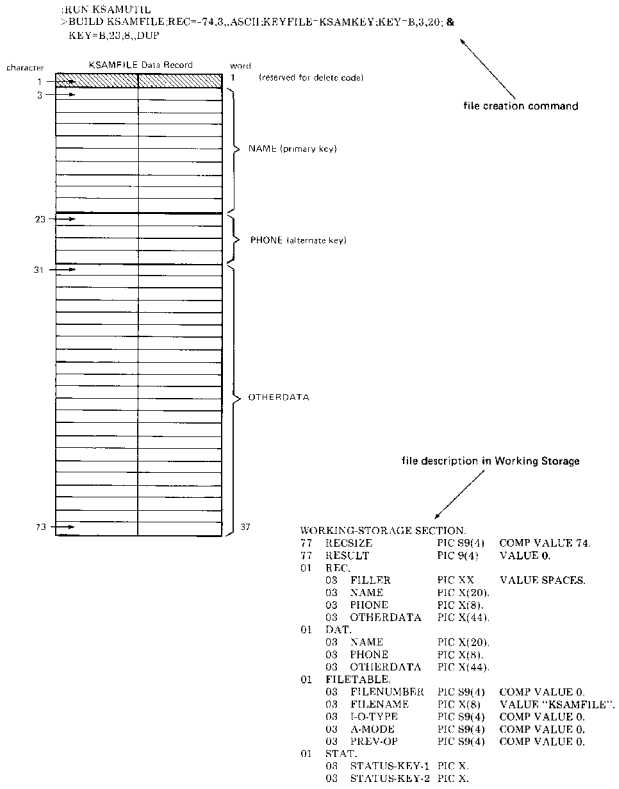


Figure 3-2. Representation of KSAMFILE Used in COBOL Examples

CKCLOSE

A call to CKCLOSE terminates file processing for the specified 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-word record containing: the name of the file, its input-output type, access mode, the filenumber 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-word (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.)

USING CKCLOSE

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.

EXAMPLES

Assuming the same *filetable* and *status* definitions used to define the sample file in Figure 3-2:

```
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 fileable, status
```

In order to logically delete records from a KSAM file, you can use the procedure CKDELETE. A logically deleted record is marked by a code of binary 1's in the first two characters of the record, 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 (described in section II) 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 (see Table 3-3).

PARAMETERS

filetable — an 8-word 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 word (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.)

USING CKDELETE

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.

SHARED ACCESS. 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.

EXAMPLES

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 3-2 and the FINISH procedure from the CKCLOSE example are assumed for these examples.

1. Sequential Delete.

   In order 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.
  .
  . <---------- check status
  .
  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.
```

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 should 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.

2. Random Delete.

   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.
    .
    . check status
    .
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 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 and a call to CKUNLOCK must follow the CKDELETE error tests and should precede the return to READ-KEY.

CKERROR

Converts 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 is 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 is 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)).

USING CKERROR

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 insures that another user cannot attempt to modify the file at the same time, and 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-word 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-word (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-word 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.

USING CKLOCK

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.

EXAMPLES

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 insure 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="0"
      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 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. You can create a KSAM file through the BUILD command of the KSAMUTIL program (refer to section II).

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-word 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 word (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. Left character is set to "0" if open is successful, to "9" if not. Right character is set to "0" if open is successful, to file system error code if not. (Refer to Status Parameter discussion earlier in this section.)

USING CKOPEN

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 words in *filetable* that specify input-output type and access mode. (Refer to Table 3-4 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 inputoutput 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.

Table 3-4.
Procedures Allowed for Input-Output Type/Access Mode Combinations

ALLOWED PROCEDURES	ACCESS MODE		INPUT-OUTPUT TYPE
CKREAD CKSTART	0 (sequential)	2 (dynamic)	0 (open for input)
CKREADBYKEY	1 (random)	2 (dynamic)	0 (open for input)
CKWRITE	0 (sequential)	2 (dynamic)	1 (open for output)
	1 (random)	2 (dynamic)	1 (open for output)
CKREAD CKSTART CKREWRITE CKDELETE	0 (sequential)	2 (dynamic)	2 (open for input/output)
CKREADBYKEY CKWRITE CKREWRITE CKDELETE	1 (random)	2 (dynamic)	2 (open for input/output)

INPUT-OUTPUT TYPE. Word 6 of *filetable* must be set to one of the following values before calling CKOPEN:

0 input only

1 output only

2 input-output

Input Only. 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.

Output Only. 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.

Input-Output. 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.

ACCESS MODE. Word 7 of *filetable* must be set to one of the following values before calling CKOPEN:

0	sequential access
1	random access
2	dynamic access

Sequential Access. With this type of 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. This type of 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. Dynamic access allows you to use any call to process a file opened for inputoutput. 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. See Random Mode discussion, above. 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.

EXAMPLES

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 re-open 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 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 with the BUILD command of program KSAMUTIL (refer to section II).

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 insure 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-word 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 word (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. Left character is set to "0" if open is successful, to "9" if not. Right character is to"0" if open is successful, to file system error code if not. (Refer to Status Parameter discussion earlier in this section.)

USING CKOPENSHR

Except that CKOPENSHR allows shared access and dynamic locking, and CKOPEN does not, a call to CKOPENSHR operates exactly like the call to CKOPEN. Upon successful execution of CKOPENSHR, the file named in *filetable* is available for the type of processing specified in *file- table*. 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 affects 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 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-word 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-word (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. (Refer to Status Parameter discussion earlier in this section.)

record — a record defined in the WORKING-STORAGE SECTION into which the contents of the next sequential KSAM record is read.

recordize — an integer (S9(4)COMP) containing the length in characters of the record being read. It must not exceed maximum record length established for the file when it was created.

USING CKREAD

The file from which the record is read must be open 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 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 and 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 insures that the correct record is modified or deleted.

SHARED ACCESS. Because CKREAD is a pointer-dependent procedure (refer to Table 3-3), 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 pointerdent 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.

EXAMPLE

Using the WORKING-STORAGE SECTION from Figure 3-2 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.

1. Example of Sequential Read

```
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.
.
.
. see CKSTART example
```

2. Example of 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-word 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 word (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. (Refer to Status Parameter discussion earlier in this section.)

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-word integer (S9(4)COMP) set to the starting character position of the key in the KSAM data record (first position is character 1). *keyloc* 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.

USING CKREADBYKEY

In order to use the CKREADBYKEY procedure, the file must be opened for either input or inputoutput. 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.

EXAMPLES

In the following examples, update information is read into the area called DAT in the WORKINGSTORAGE SECTION. (Note that in this as in the preceding examples, the WORKING-STORAGE SECTION from Figure 3-2 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.

1. 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 STATUS = "00" THEN
```

```
        DISPLAY "RECORD FOUND", REC
        GO TO FIND-RECORD
    IF STATUS = "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-word 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. (Refer to Filetable parameter discussion earlier in this section.)

status — one word (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 creation.

USING CKREWRITE

In order to call procedure CKREWRITE, the file must be open for both input and output (inputoutput 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.

REWRITE IN SEQUENTIAL MODE. 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 insure 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.

Rewriting Records With Duplicate Keys. 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 key file. 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 which is duplicated, it is good practice to use CKDELETE to delete the selected record and then rewrite it as a new record with CKWRITE.

REWRITE IN RANDOM MODE. 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 the "23".

A call to CKREWRITE in random mode only updates the first record with a key in the chain of duplicate keys.

POSITION OF POINTER. 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,

REWRITE WITH SHARED ACCESS. 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 insure 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 insures that no other users change the position of the pointer while you are sequentially updating the file.

INVALID KEY. 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.

EXAMPLES

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 3-2 and the FINISH procedure from CKCLOSE.

1. Sequential Update.

   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:

```
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.          \
77 RELOP          PICS9(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.
```

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.

---

2. Random Update. Find the record with the primary key "ECKSTEIN, LEO "and change 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 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-word 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 word (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-word 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-word 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-word 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*.

USING CKSTART

When CKSTART is executed, the key file 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 file 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".

SHARED ACCESS. 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 key file so that the position of the pointer becomes invalid for any subsequent call to a procedure that depends on the pointer position. (Refer to Table 3-3 for a list of the pointer-dependent procedures.)

EXAMPLES

Four new items must be added to the WORKING-STORAGE SECTION in Figure 3-2; 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 access mode is sequential. It also assumes the FINISH procedure from the CKCLOSE example.

1. Position by alternate key sequence:

```
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 STATUS = "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.

2. Using a Generic Key

```
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 STATUS = "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 only be done by calling CKOPENSHR to open the file, not CKOPEN.

PARAMETERS

filetable — an 8-word 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-word (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 to"00" if the file was unlocked successfully; to "31 "if the file was not locked; or to "9n" where n is a binary file system error code if the call fails for any other reason. (Refer to Status Parameter discussion earlier in this section.)

USING CKUNLOCK

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."

EXAMPLES

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 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-word 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. (Refer to Filetable Parameter discussion earlier in this section.)

status — one-word (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. (Refer to Status Parameter discussion earlier in this section.)

record — a record defined in the WORKING-STORAGE SECTION containing data to be written to the file by CKWRITE.

recordsize — an interger (S9(4)COMP) containing the length in characters of the record to be written.lt 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.

USING CKWRITE

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.

WRITING IN SEQUENTIAL MODE. 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 insures 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.

WRITING IN RANDOM MODE. 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.

WRITING WHEN ACCESS IS SHARED. 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.

INVALID KEY. 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".

EXAMPLES

Assume a KSAM file called KSAMFILE with records containing 74 characters (72 characters of data following two characters reserved for the delete code), 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 3-2 for a diagram of the structure of this file.)

The first example writes data to KSAMFILE in sequential order by the primary key. The second example, using the same DATA DIVISION and the same FINISH procedure, writes one record to the file containing the "ADAMSON JOHN" as its primary key value.

1. Example of Sequential Write.

```
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.
```

2. Example of random write.

```
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 WRITE-REC;
      ELSE GO TO FIND-REC.
WRITE-REC.
  MOVE CORRESPONDING DAT TO REC.
  CALL "CKWRITE" USING FILETABLE, STAT, REC, RECSIZE.
  IF STATUS-KEY-1="0" 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 FROM COBOL PROGRAM

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 by the KSAMUTIL >BUILD command 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.

EXAMP1. 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 STEEVENS 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 LEBERTY 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). 1
```

Figure 3-3. Sequential Write Using COBOL

```
002800  01  FILETABLE.
002900      02  FILENUMBER   PIC S9(4)   COMP VALUE 0.
003000      02  FILENAME     PIC X(8)   VALUE "GKSAMFIL".
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
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 STEEVENS 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 LEBERTY 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
```

Figure 3-3. Sequential Write Using COBOL(continued)

EXAMP2. 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 EXAM2:

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 "GKSAMFIL".
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
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 = "O" 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.
```

Figure 3-4. Sequential Read Using COBOL

```
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.
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.
```

Figure 3-4. Sequential Read Using COBOL (continued)

```
Outtput 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 LEBERTY 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

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 LEBERTY 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
```

Figure 3-4. Sequential Read Using COBOL (continued)

EXAMP3. 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 in 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 "GKSAMFIL".
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 3-5. 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                 "1  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, "1 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.
```

Figure 3-5. 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.
```

Figure 3-5. Random Update with COBOL (continued)

```
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.


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 PENNSYLVANIS     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 = 231  KEY = TURNEW    IVAN
CKREADBYKEY  ERROR, STATUS = 231  KEY = FORD      GERALD
CKWRITE  ERROR, STATUS = 22
WESTER     ELDER     287-44598 1256 KINGFISHER ST.  SUNNYVALE   CA. 9430A
```

Figure 3-5. Random Update with COBOL (continued)

Note

Note that the input contains data that results in error messages. The name IVAN TURNEWR 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.

4

USING KSAM FILES IN SPL PROGRAMS

KSAM FILE SYSTEM INTRINSICS

The Multi-Programming Executive Operating System (MPE) provides a set of procedures, known as intrinsics. A subset of these intrinsics makes up the file system, a set of procedures used to manipulate files. KSAM files are processed using these same intrinsics with the following exceptions: seven new intrinsics are added for KSAM files, and four of the file system intrinsics do not apply to KSAM files. (Refer to Table 4-1 for a list of the KSAM file system intrinsics.)

Table 4-1. KSAM File System Intrinsics

INTRINSIC NAME	KSAM ONLY	NOT USED BY KSAM	DIFFERENCES IN FORMAT	FUNCTION
FOPEN			*ksamparam* replaces *formmsg* as sixth parameter.	Opens a KSAM file for access and assign file number to file.
FCLOSE			none	Closes a KSAM file to further access.
[FRENAME]		X	—	If called for KSAM file, returns CCL error code.
FREAD			none	Reads next record in sequential order by key.
*FREADC	X		all new	Reads next record in chronological sequence.
*FREADBYKEY	X		all new	Reads record identified by key value.
FREADDIR			none	Reads record identified by chronological position.
[FREADSEEK]		X	If called for KSAM file, returns CCL error code.	
FWRITE			*control* parameter included for compatibility only.	Writes record to KSAM file.
[FWRITEDIR]		X	—	If called for KSAM file, returns CCL error code.
*FREMOVE	X		all new	Deletes current record from KSAM file.
FUPDATE			none	Updates last referenced record.
FSPACE			none	Spaces forward or backward in file.
*FFINDBYKEY	X		all new	Positions current record pointer to record located by key value.

Table 4-1. KSAM File System Intrinsics (continued)

INTRINSIC NAME	KSAM ONLY	NOT USED BY KSAM	DIFFERENCES IN FORMAT	FUNCTION
*FFINDN	X		all new	Positions current record pointer to relative record number in key sequence.
FPOINT			none	Positions current record pointer to relative record number in chronological sequence.
FGETINFO			none	Requests file access and status information.
*FGETKEYINFO	X		all new	Requests access and status information on KSAM file.
[FRELATE]		X	—	If called for KSAM file, returns CCE and false condition.
FCHECK			none	Requests details of file input/output errors.
FERRMSG			none	Prints message corresponding to FCHECK error code.
FCONTROL			*param* parameter included for compatibility only	Ensures that input/output is complete or positions to first sequential record by key value; other options not available for KSAM file.
FSETMODE			none	Verifies critical output as part of write operation; other options not available for KSAM file.
FLOCK			none	Dynamically locks file.
FUNLOCK			none	Dynamically unlocks file.
FREADLABEL			none	Reads user's file label.
FWRITELABEL			none	Writes user's file label.

KSAM File System Intrinsics (continued)

INTRINSIC NAME	KSAM ONLY	NOT USED BY KSAM	DIFFERENCES IN FORMAT	FUNCTION
*HP32208	X		all new	Identifies the KSAM version.
FDELETE		X		Deactivates a RIO record.
FDEVICE-CONTROL		X		Provides control operations to a printer, Workstation Configurator, or a spooled device.
FFILEINFO			none	Provides access to file information.
FLABELINFO			The information returned may not match what is returned from FFILEINFO.	Returns information from the file label of a disc file.
FPARSE			none	Passes and validates file designators.
FREAD-BACKWARD		X		Reads a logical record backward from the current record pointer. Data is presented to the user as if read forward.

CALLING INTRINSICS FROM SPL

An intrinsic used in an SPL program must be declared at the beginning of the program following all other declarations. There are two ways to declare an intrinsic: one is to make an external procedure declaration, and the other is to use the INTRINSIC declaration. Since declaring an external procedure is a long process, you can save space and time by using the INTRINSIC declaration as follows:

`INTRINSIC` *intrinsicname*`,` *intrinsicname*`,...,` *intrinsicname*`;`

You name all the intrinsics used in your program in the *intrinsicname* list. When more than one intrinsic is named, the names must be separated by commas.

You call an intrinsic by writing the intrinsic name followed by a list of parameters enclosed in parentheses. These parameters must be in the order established for each intrinsic as shown in the intrinsic formats later in this section. Every parameter that is specified as a variable or an array must be declared before the intrinsic is called. The formats that describe intrinsics define the variable or array type of each parameter; specify whether it can be passed by value or must be passed by reference; and indicate whether any parameters are optional and if so which ones.

In summary, to call an intrinsic from an SPL program:

1. Refer to the intrinsic format to determine the parameter type and position.

2. Declare any variable or array names to be passed as parameters at the beginning of the program.
3. Declare the intrinsic name in an INTRINSIC statement.
4. Issue the intrinsic call where appropriate in your program.

KSAM INTRINSIC SUMMARY

Table 4-1 is provided to give an overview of the intrinsics available for accessing KSAM files. In this table, the intrinsics are organized into functional groupings. In the body of this section, however, the intrinsic descriptions are in alphabetic order so that they may be referenced easily.

In Table 4-1, an asterisk (*) preceding an intrinsic name indicates that this intrinsic applies only to KSAM files. A bracket around an intrinsic name indicates that the intrinsic should not be used for KSAM files.

INTRINSIC FORMAT

Intrinsic format is illustrated below using FCHECK as an example.

```
          IV        I         I      D       I       O-V
FCHECK(filenum,errorcode,tlog,blknum,numrec);
```

Optional parameters are indicated by an underline under each option and by the superscript O-V. The parameter type and whether it is passed by value is shown by the superscript over each parameter. Possible parameter types are:

BA	Byte array
BP	Byte pointer
D	Double
DA	Double array
DV	Double by value
I	Integer
IA	Integer array
IV	Integer by value
L	Logical
LA	Logical array
LV	Logical by value
R	Real

PASSING PARAMETERS

Integer,logical and double type parameters can be passed by value. This means that the actual value can be specified in the intrinsic call instead of a variable or array name. When a parameter is passed by reference (default for all parameter types), the address in the caller's data area of the named variable or array is made available to the intrinsic. If the variable or array is modified by execution of the intrinsic, the storage in the caller's data area is updated. When a parameter is passed by value, the corresponding variable in the calling routine is unchanged.

OPTIONAL PARAMETERS

If any parameters can be omitted, the superscripts that describe individual parameters are followed by the superscript O-V, option variable. O-V means that at least one parameter in the list is optional. Since all parameters are recognized by their position in the list, a parameter may be omitted but its preceding comma must be included. If one or more parameters are omitted from the end of the list, this is indicated by placing the terminating parenthesis after the last specified parameter.

For example:

`FCHECK(FILEX , , , , REC)`	*only the first and fifth parameters are included*
`FCHECK(2,ERR)`	*the last three parameters are omitted; note that filenum is passed by value*

KSAM RECORD POINTERS

Certain KSAM procedures use pointers that indicate the current record position in the file. Depending on the procedure, either of two pointers may be used:

* Logical Record Pointer	Points to a key in the key file that identifies a particular record in the data file.
* Chronological Record Pointer	Points directly to a record in the data file based on its chronological record number.

Procedures that use these pointers are either *pointer-dependent* or *pointer-independent*. Pointerdependent procedures expect the pointer to be positioned in order to execute correctly. Pointerindependent procedures, on the other hand, execute regardless of where the pointer is positioned, and in most cases, they position the pointer. Because the position of the pointer is significant for pointer-dependent procedures, Table 4-2 defines exactly where each pointer is located following successful execution of those procedures that either depend on or position the pointer.

Table 4-2. Positioning the Pointers

PROCEDURE NAME	POINTER TYPE	POINTER-DEPENDENT	POSITION OF POINTER AFTER EXECUTION OF PROCEDURE
FFINDBYKEY	Logical	NO	Points to key whose value was specified in call.
FFINDN	Logical	NO	Points to key whose relative record number was specified in call.
FREADBYKEY	Logical	NO	Points to key whose value was specified in call.
FWRITE	Logical	NO	Points to key whose value is next in ascending key sequence to the key value in the record just written.
FPOINT	Chrono-logical	NO	Points to record whose relative record number was specified in call.
FREADDIR*	Chrono-logical	NO	Points to record whose relative record number was specified in call.
FREAD	Logical	YES	Pointer remains positioned to key for the record just read; *unless* next call is to FREAD or to FUPDATE followed by FREAD, in which case, pointer is advanced to next key in sequence before the next FREAD reads the record. (This permits sequential reads and updates.)
FSPACE	Logical	YES	Positioned forward or backward, in key sequence, the number of records specified in call.
FREMOVE	Logical	YES	Points to next key value, in ascending sequence, to the key value in the record just deleted.
FUPDATE	Logical	YES	Pointer remains positioned to key of the record just modified; *unless* any key value is changed, in which case, it points to next key in ascending sequence after the key in the modified record.
FREADC	Chrono-logical	YES	Pointer remains positioned to the record just read; *unless* next call is to FREADC, in which case, it points to next record in ascending chronological sequence.

* Except for FREADDIR, each of these procedures positions both pointers. That is, all procedures that position the logical pointer also position the chronological pointer, and all calls (except FREADDIR) that position the the chronological poiner also position the logical pointer.

(Refer to appendix B, Extra Data Segments With Shared Access, for details of how KSAM determines pointer position.)

SHARED ACCESS

The position of the record pointers is crucial during shared access because the pointers are maintained in separate control blocks (extra data segments) for each open file. Thus, if the same file is opened by different users, any user may change the key file structure by adding or deleting records so that other users' pointers become invalid. To avoid this problem, it is good practice to lock the file in a shared environment before calling a procedure that positions the pointer and leave the file locked until any pointer-dependent operation is complete. This means that you should lock the file, call a procedure that sets the pointer, and then call a procedure that reads the file sequentially or updates the file, and then unlock the file so other users may access it. Once the file is unlocked, no user should assume that his pointers will still be valid. Before using a pointer again, it must be re-established.

FCHECK

INTRINSIC NUMBER 10

Requests details about file input/output errors.

```
                IV        I     I       D       I     0-V
  FCHECK(filenum,errorcode,tlog,blknum,numrecs);
```

When a file intrinsic returns a condition code indicating a physical input/output error, additional details may be obtained by calling FCHECK. This intrinsic applies to files on any device.

FCHECK accepts zero as a legal *filenum* parameter value. When zero is specified, the information returned in *errorcode* reflects the status of the last call to FOPEN. When an FOPEN fails, there is no file number that can be referenced in *filenum*. Therefore, when an FOPEN fails, a *filenum* of zero can be used in the FCHECK intrinsic call to obtain the *errorcode* only. If the *tlog*, *blknum*, or *numrecs* parameters are specified, a zero value is returned to these parameters. If a *filenum* of zero is used for a file which has been opened but not yet closed, the returned *errorcode* is meaningless.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file for which error information is to be returned.

errorcode *integer (optional)*

A word to which is returned the error code specifying the type of error that occurred. If no error occurred *errorcode* is set to zero. (Refer to Table 4-3 for the *errorcode* values.) The intrinsic FERRMSG returns a displayable message that corresponds to the value of *errorcode*.

Default: The error code is not returned.

tlog *integer (optional)*

A word to which is returned the transmission log value recorded when an erroneous data transfer occurs. This word specifies the number of words not read or written (those left over) as the result of an input/ output error.

Default: The transmission log value is not returned.

blknum *double (optional)*

A double word to which is returned the relative number of the block involved in the error.

Default: The block number is not returned.

numrecs *integer (optional)*

A word to which is returned the number of logical records in the bad block.

Default: The number of logical records is not returned.

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because *filenum* was invalid and errorcode is 72, or a bounds violation occurred while processing this request and *errorcode* is 73.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

Table 4-3. FCHECK *errorcode* Parameter Format

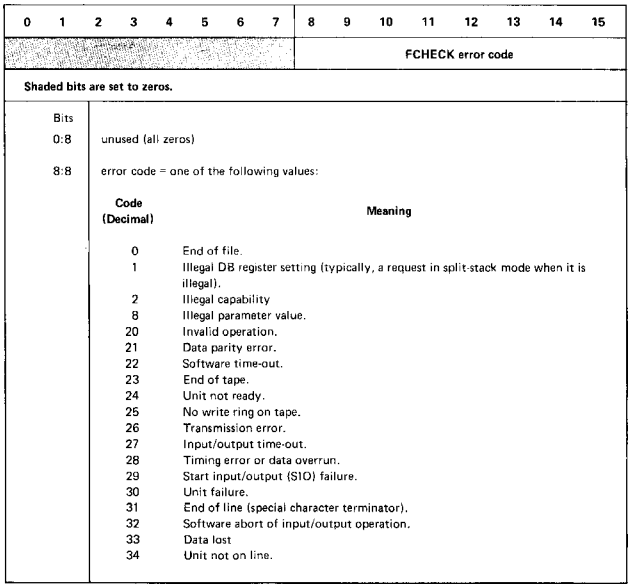


Table 4-3. FCHECK *errorcode* Parameter Format (continued)

	Code (Decimal)	Meaning
	35	Data set not ready.
	36	Invalid disc address.
	37	Invalid memory address.
	38	Tape parity error.
	39	Recovered tape error.
	40	Operation inconsistent with access type.
	41	Operation inconsistent with record type.
	42	Operation inconsistent with device type.
	43	The *tcount* parameter value exceeded the *recsize* parameter, but the *multi-record access aoption* was not specified when the file was opened.
	44	The FUPDATE intrinsic was called, but the file was positioned at record zero. (FUPDATE must reference the last record read, but no previous record was read.)
	45	Privileged file violation.
	46	File space on all discs in the device class specified is insufficient to satisfy this request.
	47	Input/output error on a file label.
	48	Invalid operation due to multiple file access.
	49	Unimplemented function.
	50	The account referenced does not exist.
	51	The group referenced does not exist.
	52	The referenced file does not exist in the system (permanent) file domain.
	53	The referenced file does not exist in the job temporary file domain.
	54	The file reference is invalid.
	55	The referenced device is not available.
	56	The device specification is invalid or undefined.
	57	Virtual memory is not sufficient for the file specified.
	58	The file was not passed (typically, a request for $OLDPASS when there is no $OLDPASS).
	59	Standard label violation.
	60	Global RIN not available.
	61	Group disc file space exceeded.
	62	Account disc file space exceeded.
	63	Non-sharable device (ND) capability required but not assigned.
	64	Multiple RIN (MR) capability required but not assigned.
	66	Plotter limit switch reached.
	67	Paper tape error.
	68	System internal error.
	69	Miscellaneous (ATTACHIO) input/output error.
	71	Too many files opened for process.
	72	Invalid file number.
	73	Bounds check violation.
	77	NO-WAIT input/output operation is pending.
	78	There is no NO-WAIT input/output for any file.
	79	There is no NO-WAIT input/output for file specified.
	80	Configured maximum number of spoolfile sectors would be exceeded by this output request.
	81	No SPOOL class defined in system.

Table 4-3. FCHECK *errorcode* Parameter Format (continued)

	Code (Decimal)	Meaning
	82	Insufficient space in SPOOL class to honor this input/output request.
	83	Extent size exceeds maximum allowable.
	84	The next extent in this spoolfile resides on a device which is unavailable to the system (i.e., the device is =DOWN).
	85	Operation inconsistent with spooling; e.g., attempt to read hardware status.
	86	Spool process internal error.
	87	Offset to data is greater than 255 sectors.
	89	Power failure.
	90	The calling process requested exclusive access to a file to which another process has access.
	91	The calling process requested access to a file to which another process has exclusive access.
	92	Lockword violation.
	93	Security violation.
	94	Creator conflict in use of FRENAME intrinsic (user is not the creator).
	95	"BROKEN" terminal read.
	96	Miscellaneous disc input/output error (device may require HP Customer Engineer attention).
	97	CONTROL Y processing requested but no CONTROL Y PIN exists.
	98	Input/output read time has overflowed.
	99	Magnetic tape error. Beginning of tape (BOT) found while requesting a backspace record (BSR) or a backspace file (BSF).
	100	Duplicate file name in the system file directory.
	101	Duplicate file name in the job temporary file directory.
	102	Directory input/output error.
	103	System directory overflow.
	104	Job temporary directory overflow.
	105	Illegal variable block structure.
	106	Extent size exceeds maximum allowable.
	107	Offset to data is greater than 255 sectors.
	108	Inaccessible file due to a bad file label.
	109	Illegal carriage control option.
	110	The intrinsic attempted to save a system file in the job temporary file directory.
	170	FPOINT intrinsic tried to position to a record that was flagged for deletion.
	171	Duplicate key value when duplicates not allowed.
	172	Key not found; no such key value.
Codes 170 – 200 Reserved for KSAM File Errors	173	*tcount* parameter larger than record size.
	174	Cannot get extra data segment for this file.
	175	KSAM internal error.
	176	Illegal extra data segment length.
	177	Too many extra data segments for this process.
	178	Not enough virtual memory for extra data segment.
	179	Undefined.
	180	Undefined.
	181	Invalid key starting position.
	182	File is empty.
	183	Record does not contain all the keys.

Table 4-3. FCHECK *errorcode* Parameter Format (continued)

	Code (Decimal)	Meaning
	184	Invalid record number in FFINDN intrinsic; record number is negative.
	185	Sequence error in primary key.
	186	Invalid key length; (numeric display and packed decimal type keys shorter than length specified at creation).
	187	Invalid key specification; keys illegal.
	188	Invalid device specification.
	189	Invalid record format.
	190	Invalid key blocking factor value.
	191	Incorrect record; as a result of a previous system failure, a key points to a record that has a different key value.
	192	System failure occurred when the KSAM file was open; run KEYINFO of KSAMUTIL to recover.
	193	Undefined.
	194	Undefined.
	195	Undefined.
	196	Undefined.
	197	Undefined.
	198	Undefined.
	199	Undefined.
	200	Undefined.

FCLOSE

INTRINSIC NUMBER 9

Closes a file.

```
                IV        IV        IV
FCLOSE(filenum,disposition,seccode);
```

The FCLOSE intrinsic terminates access to a file. This intrinsic applies to files on all devices. FCLOSE deletes the buffers and control blocks through which the user process accessed the file. It also deallocates the device on which the file resides and it may change the disposition of the file. If you do not issue FCLOSE calls for all files opened by your process, such calls are issued automatically by MPE when the process terminates.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be closed.

disposition *integer by value (required)*

Indicates the disposition of the file, significant only for files on disc. This disposition can be overridden by a corresponding parameter in a :FILE command entered prior to program execution. The disposition options are defined in Table 4-4

Default: disposition is zero for no change, no return of disc space.

seccode *integer by value (required)*

Denotes the type of security initially applied to the file, significant only for new permanent files. The options are:

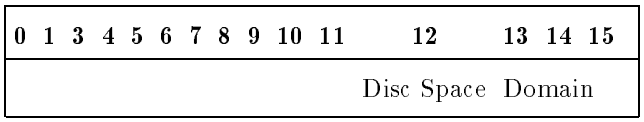
0—Unrestricted access—the file can be accessed by any user, unless prohibited by current MPE provisions.

1—Private file creator security—the file can be accessed only by its creator.

Default: seccode is zero for unrestricted access.

Note Both parameters are required when FCLOSE is specified in a program; the default values are used when MPE closes any open files at the end of a job or session.

Table 4-4. FCLOSE *disposition* Parameter Bit Settings



Set shaded areas to zero for KSAM files.

BITS	OPTION	SETTINGS
13:3	Domain	000 = No change. (default) The disposition code remains as it was before the file was opened. Thus, if the file is new, it is deleted by FLCOSE: otherwise, the file is assigned the domain to which it previously belonged.
		001 = Permanent file. If a disc file, it is saved in the system domain. If the file is a new or temporary file, an entry is created for it in the system file directory. An error code is returned if a file of the same name already exists in the directory. This disposition has no effect when the file is an old permanent file on disc.
		010 = Temporary job file. This file is retained in the user's temporary (job/session) file domain. It can be requested by any process within the job/session. The uniqueness of the file name is checked and if a file of the same name already exists, an error code is returned.
		011 = Temporary job file. This option has the same effect as disposition code 010.
		100 = Released file. The file is deleted from the system.
12:1	Disc Space Disposition	0 = No return. (default) Any disc space allocated to the file that is beyond the end-of-file indicator is not returned to the system. *This option is recommended for KSAM files.*
		1 = Return disc space. Any disc space allocated beyond the end-of-file indicator is returned to the system. *This option is not recommended for KSAM users since the returned space cannot be recovered.*

CONDITION CODES

CCE The file was closed successfully.

CCG Not returned by this intrinsic.

CCL The file was not closed, perhaps because an incorrect *filenum* was specified, or because another file with the same name and disposition exists in the system.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FCLOSE

The FCLOSE intrinsic terminates access to a file so that it cannot be accessed again by the current program until it is re-opened. FCLOSE can also be used to change the disposition currently assigned to a file by a previous FOPEN.

Because of the special structure of KSAM files, it is not good practice to set the disc space bit in the *disposition* parameter in an attempt to save disc space. For this reason, 4 is the largest value that should be assigned to the *disposition* parameter when using FCLOSE to close a KSAM file.

When a file is opened by the FOPEN intrinsic, a file count maintained by the system is incremented by one. When the file is closed with FCLOSE, the file count is decremented by one. If more than one FOPEN is in effect for a particular file, its disposition is recorded by the FCLOSE call but is not put into effect until the file count is zero. The effective disposition at that time is the smallest non-zero disposition parameter specified among all the FCLOSE calls issued against the file. For example, a file XYZ is opened three successive times by a process. The first FCLOSE disposition is 1, the second FCLOSE disposition is 4, and the third and last FCLOSE disposition is 2. The final disposition of the file XYZ is 1, that is, it is saved as a permanent file with no return of disc space.

The use of FCLOSE differs slightly in its application to new files or to existing files.

CLOSING A NEW KSAM FILE

When a new file is created by FOPEN, the job temporary and system file domains are not searched to determine whether a file of the same name exists already. Only when a file is closed and saved as a permanent or temporary file with FCLOSE, is such a search conducted. The job temporary file domain is searched if the file is closed with the domain field of *disposition* set to 2 or 3 (save as temporary file); the system file domain is searched if the file is closed with domain set to 1 (save as permanent file). If a file of the same name is found in either directory, an error code is returned. Thus it is possible to open a new file with the same name as an existing file, but an error results if FCLOSE is used to save such a file in the same domain with a file of the same name.

In general, unless you plan to use the file once and then delete it, a newly created file should be closed using FCLOSE with the disposition parameter set to 1, 2, or 3. There is no need to set *disposition* to 4 in order to delete a new file since a new file is deleted when it is closed with a *disposition* of 0.

The security code parameter (*seccode*) is set only when the *disposition* parameter is set to 1. If you want exclusive access to a file being saved as a new permanent file, you should set *seccode* to 1 when you close the file for the first time. Otherwise, the file can be accessed by any other user.

In Figure 4-1, a new file is closed and saved as a permanent file in the system file domain (*disposition* = 1), and access is permitted to the file by other users (*seccode* = 0).

CLOSING AN EXISTING KSAM FILE

Unless you plan to change the domain where a file is saved, you usually close an existing file with both FCLOSE parameters set to zero. There are two limitations: you cannot change an existing permanent file to a temporary file, and you cannot change the security code that was assigned to a permanent file at creation.

```
<<*************************************>>
<<* READ DATA FROM $STDIN DEVICE       *>>
<<*************************************>>
L1 :
READ(INPUT,-72): <<READ ONE RECORD FROM $STDIN>>
IF >
THEN BEGIN <<END OF FILE ON $STDIN>>
  FCLOSE(FILNUM,1,0); <<CLOSE THE KSAM FILE>>
  IF <> THEN
    BEGIN <<CANNOT CLOSE THE KSAM FILE>>
       MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE":
       PRINT (MESSAGE,-29,0):
       FCHECK(FILNUM,ERRORCODE): <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH):<<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0)1 <<PRINT ERROR MESSAGE>>
    END;
  TERMINATE;
 END;
IF <
THEN BEGIN
  MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
  PRINT(MESSAGE,-34,0);
  TERMINATE;
 END;
PRINT(OUTPUT,-72,0); <<ECHO CHECK>>
```

Figure 4-1. FCLOSE Example

Assume, for example, that a file was closed as a job temporary file. Should you want to make the file permanent, close the file with the following call:

```
FCLOSE(FILENUM,1,0)  close job temporary file as permanent file
```

If, however, you want to maintain this file with its cuff ent disposition, you would close it with the following call:

```
FCLOSE(FILENUM,0,0)  close file with current disposition
```

Regardless of the value assigned to *seccode*, the type of security initially applied to the file when it is closed as a new permanent file is not subsequently changed.

DELETING A KSAM FILE

The FCLOSE intrinsic can be used to delete a KSAM file from the system. If you intend to use a new file once only, you can delete it at the same time you close it for the first time by setting the FCLOSE parameters to zero:

```
FCLOSE(FILNUM,0,0)  delete a new file
```

In this case, because disposition is zero, the file is returned to its domain before FCLOSE is executed. Since the file is not assigned a domain until it is closed the first time, this effectively deletes the file.

A file that has been assigned to a domain by a previous FCLOSE, can be deleted by the call:

```
FCLOSE(FILNUM,4,0)  delete an existing file
```

Note that the only other methods for deleting a KSAM file are to use the KSAMUTIL>PURGE command, or to issue two MPE :PURGE commands, one for the data file and one for the key file.

FCONTROL

INTRINSIC NUMBER 13

Performs control operations on a KSAM file.

```
                IV          IV          L
FCONTROL(filenum,controlcode,param);
```

The FCONTROL intrinsic performs various control operations on a KSAM file. When specified for a KSAM file, these control operations are limited to the following:

- Complete input/output
- Rewind the file

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file for which the control operation is to be performed.

controlcode *integer by value (required)*

An integer identifying the operation to be performed:

2 = Complete Output. This insures that requested output has been physically completed; that is, that the key buffers, data buffers, and KSAM control information are all written to disc.

When access is shared, you must lock the file with FLOCK before calling FCONTROL with control code 2.

5 = Rewind File. This repositions the file at its beginning, so that the next record read or written is the first logical record in the file. When this code is used for KSAM files, the file is *not* repositioned to the first physical record but to the first logical record. The first logical record is the record with the lowest value in the current key (primary or alternate)

[***text removed at this point***]

param *logical (required)*

This parameter may be specified as any variable or word identifier; it is needed by FCONTROL to satisfy internal requirements of the intrinsic, but serves no other purpose and is not modified by the instrinsic.

CONDITION CODES

CCE Request granted.

CCG Not returned by this intrinsic.

CCL Request denied because an error occurred. Returned if any control code other than 2, 5, 6, or 7 is specified for a KSAM file; or the file was opened for shared access, but was not locked for control code 2 or 6.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FCONTROL

FCONTROL provides four control functions for KSAM files. These allow you to write the key and data buffers and all KSAM control information to disc; to position the logical record pointer to the first logical record in the file; to write the buffers, KSAM control information, *plus* the MPE endof-file and the latest extent bit map, to disc; and to clear all the data buffers and the latest control information from disc.

The control functions that write the buffers to disc (2 and 6) require that you lock the file before calling them in a shared access environment.

USING CONTROL CODE 2

When you use control code 2, the data block and key block buffers and the KSAM control information (including the KSAM end-of-file) are written to disc. (The data written is that contained in the Extra Data Segment for the open file—refer to Figure B-11 for details.) This control code is particularly useful to make sure the KSAM file reflects current changes. Suppose, for instance, that you open a KSAM file exclusively for a long period of time and that your data buffer holds many records. In this case, you can call FCONTROL with code 2 after writing or updating a certain number of records to insure that no more than that number of records will be lost in case of a system failure.

For example, you could call FCONTROL every 10 records:

```
  IF COUNT = 10              counter set by each FWRITE or FUPDATE
  THEN BEGIN
 FCONTROL(FILNUM,2,DUMMY);
END;
```

Note Note that the parameter DUMMY has no function. It is supplied because all FCONTROL parameters are required. It should be declared in the program as a word variable: LOGICAL DUMMY;

As a result of the call shown above, you can never lose more than 10 records in case of a system failure. When a system failure occurs with a KSAM file open, you must run the KSAMUTIL command KEYINFO to allow the file to be reopened. KEYINFO also sets the MPE end-of-file to the current position of the KSAM end-of-file. Control code 2 of FCONTROL makes sure that the KSAM end-of-file follows the last record written to your file.

In a shared environment, be sure to lock your file before calling FCONTROL with control code 2. Otherwise, the call will fail.

USING CONTROL CODE 5

This control code repositions the file to the first logical record, that is, the record with the lowest key value. The key that determines this position can be the primary key or an alternate key, depending on which key was accessed last. Suppose you want to read the KSAM file in sequence starting with the record containing the lowest primary key value, you can position to this record using FCONTROL as follows:

```
FCONTROL(FILNUM,5,DUMMY);   positions to 1st record in primary key sequence
```

USING CONTROL CODE 6

This control code performs the same functions as control code 2, except that it also writes the MPE end-of-files for the KSAM files and the latest extent bit map to disc. Because it must access the MPE control blocks as well as the KSAM control block, this code takes more time than code 2. Also, since the MPE end-of-files and the extent bit map are written to disc automatically whenever a new extent is allocated, this code is useful primarily when a series of updates changes the buffers but does not cause new extents to be allocated, and when access to the file is exclusive. *If access is shared, you must lock the file before using control code 6.*

USING CONTROL CODE 7

This control code clears the buffers so that the next call to a read instrinsic must get the record from disc rather than from the buffers. It also forces the latest control information to be read from disc to the buffers. Note that a call to FLOCK will also clear the buffers. The advantage of FCONTROL with code 7 over FLOCK is that it saves time—the buffers are cleared without locking and then unlocking the file. Thus, you can call FCONTROL with code 7 immediately before calling a read instrinsic in a shared environment in order to get the latest information from disc. However, this does not guarantee that this latest information is not changed (modified or deleted) by other users while you are calling FCONTROL. The only complete safeguard is to lock the file before the read. In any case, if you are making modifications, you should lock the file. For example:

```
FCONTROL(FILNUM,7,DUMMY);     clear buffers
FREAD(FILNUM,DATA,-72);       read record from file
⋮                             lock file
FLOCK(FILNUM,TRUE);
FREAD(FILNUM,DATA,-72);       rewrite record just read
FUPDATE(FILNUM,DATA,-72);
FUNLOCK(FILNUM);              unlock file
```

FERRMSG

INTRINSIC NUMBER 307

Returns message corresponding to FCHECK error number.

```
                I       LA       I
FERRMSG(errorcode,msgbuf,msglgth);
```

A call to FERRMSG causes a message to be returned to *msgbuf* that corresponds to an FCHECK error or number. This makes it possible to display an error message from your program. The message describes the error associated with the error number provided in the parameter *errorcode*.

PARAMETERS

errorcode *integer (required)*

A word identifier containing the error code for which a message is tobe returned. It should contain an error number returned by FCHECK.

msgbuf *logical array (required)*

A logical array to which the message associated with *errorcode* is returned by FERRMSG. In order to contain the message string, *msgbuf* must be defined as at least 72 characters (36 words) long.

msglgth *integer (required)*

A word identifier to which is returned the length of the *msgbuf* string. The length is returned as a positive byte count.

CONDITION CODES

Condition codes are not returned by this procedure.

USING FERRMSG

This intrinsic is called usually following a call to FCHECK. The error code returned in the call to FCHECK can then be used as a parameter in the call to FERRMSG.

For example, suppose a CCL condition is returned by a call to FCLOSE, a call to FCHECK requests the particular eff or code, then a call to FERRMSG can be used to retrieve a printable message associated with the code.

```
      FCLOSE(FILNUM,1,0);
      IF<
      THEN BEGIN
   FCHECK(FILNUM,ERRNUM);
   FERRMSG(ERRNUM,MESSAGE,LENGTH);
   PRINT(MESSAGE,-LENGTH,0);
  END
      TERMINATE;
```

The message printed explains the FCHECK code. If the FCHECK code has no assigned meaning, the following message is returned:

```
UNDEFINED ERROR errorcode
```

FFINDBYKEY

INTRINSIC NUMBER 302

Positions record pointer to record located by a key value.

```
                  IV       BA          IV          IV         IV
FFINDBYKEY(filenum,keyvalue,keylocation,keylength,relop);
```

When FFINDBYKEY is executed, the logical record pointer is set to the beginning of a record located by this intrinsic. The particular key is defined by the *keylocation* parameter, The pointer is positioned to the first record containing a key value that bears the relation specified by *relop* to the value specified by *keyvalue*. A partial key can be specified by a *keylength* value less than the defined key length. If, however, the key type specified at file creation was numeric display or packed decimal, a type where the sign is stored in the least significant byte, partial keys cannot be specified.

FFINDBYKEY also positions the chronological pointer.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be positioned.

keyvalue *byte array (required)*

A byte array containing a value that is used to locate the record at which the pointer is positioned. The key value in the record must be in the relation specified by *relop* to the value in array *keyvalue*.

keylocation *integer by value (required)*

The *keylocation* parameter specifies the relative byte location of the key being used. Bytes are numbered starting with 1. If *keylocation* is zero, then the primary key is used.

keylength *integer by value (required)*

This parameter specifies the length of the key in bytes. If it equals zero, the entire key is used. If less than the full key length (generic key), then only the length specified here is used in the comparison with *relop*. The *keylength* parameter must be equal to or less than the full length of the key when the file was created. For keys of type numeric display or packed decimal, the full key length must be used.

relop *integer by value (required)*

A relational operator that specifies the relation of the key value in the file to the value specified in *keyvalue*. The record to which the file is positioned will have this relation to *keyvalue* following execution of FFINDBYKEY:

0 - equal
1 - greater than
2 - equal to or greater than

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

CONDITION CODES

CCE	Request granted.
CCG	The requested position was beyond the logical end-of-file or beginning of file.
CCL	Request denied because an error occurred. The error could be a disc input/output error; the relational operator (*relop*) could not be satisfied; a *keylength* less than the fulllength was specified for a key with numeric display or packed decimal format; or a key is not found in the key file when the relational operator is equal.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FFINDBYKEY

The intrinsic FFINDBYKEY allows you to position the file to a record containing a particular key value. Usually, you will do this in order to read in ascending sequence from that particular record. If you simply want to locate and read a single record, you should use FREADBYKEY.

In Figure 4-2, FFINDBYKEY is used to position the file to the record containing the lowest value of an 8-byte alternate key in which a telephone number is stored. After FFINDBYKEY positions the file to this record, a series of FREAD statements read the records in ascending order according to the value of the key specified by FFINDBYKEY. (Refer to shaded portions of the program for the FFINDBYKEY declarations and statements).

FFINDBYKEY can also be used prior to a call to FREADC in order to position the chronological pointer to the record located by the specified key.

USING APPROXIMATE KEYS

In order to find the lowest-valued telephone number, *keyvalue* is set to the value "000-0000". The key to be searched for this value is identified by its position in the record. In this case, the alternate key containing the telephone number starts in byte position 21, and *keylocation* is set to the value 21. The fulllength of the key is specified in *keylength* as the value 8. In order to position to the record whose alternate key value is equal to or greater than "000-0000", the value of *relop* is set to 2.

When executed, FFINDBYKEY will locate the record with an 8-byte value starting in byte 21 that is either equ"000-0000" or is the lowest value greater "000-0000". Since the value "000-0000" is not a valid telephone number, the value of *relop* could be set to 1 indicating the lowest value greater than "000-0000". An error condition is returned if the value in *keyvalue* cannot be located. For this reason, *relop* should not be set to 0 unless it is expected that the value being sought exists.

USING PARTIAL (GENERIC) KEYS

If the value of *keylength* is less than the length of the key at creation, this allows a search for a partial (generic) key. For example, assume a file with a 20- byte key starting in byte 2 of each record. This key contains a name entered last name first. If you want to find and read

all records starting with the letter "R" through the last record in sequence by key, you could assign the following FFINDBYKEY values:

```
INTEGER       FILNUM;
BYTE ARRAY    FILNAME(0:9):="KSAMFILE ";
BYTE ARRAY    KEYVALUE(0:4):="R";
INTEGER       KEYLENGTH:=1;
INTEGER       KEYLOCATION:=2;
INTEGER       RELOP:=2;
      :
INTRINSIC FOPEN,FCLOSE,FREAD,FWRITE,FFINDBYKEY;
      :
FFINDBYKEY(FILNUM,KEYVALUE,KEYLOCATION,KEYLENGTH,RELOP)
```

When executed, FFINDBYKEY will position to the first record with a key value whose first (leftmost) character is the letter "R". A subsequent series of FREADs will read that record and position to the next record in sequence by the same key.

SHARED ACCESS

If you use FFINDBYKEY to position the pointer before calling another procedure to read or update the file in a shared environment, you must call FLOCK to lock the file before calling FFINDBYKEY. Then, after performing the read or update operation, you can unlock the file. If you call FFINDBYKEY and then lock the file before an operation that depends on the record pointer, another user could move the pointer between the call to FFINDBYKEY and FLOCK.

```
<<*******************************************************>>
<<*                                                     *>>
<<*              EXAMPLE 2                              *>>
<<*       READ A KSAM FILE SEQUENTIALLY                 *>>
<<*                                                     *>>
<<*******************************************************>>
       INTEGER FILNUM:
       INTEGER ERRORCODE,LENGTH;
       BYTE ARRAY FILNAME(0:9)="JEXAMFIL ";
       ARRAY MESSAGE(0:35);
       ARRAY INPUT(0:39);
       ARRAY OUTPUT(*)=INPUT;
       BYTE ARRAY KEYVALUE(0:7):="000-0000";
       INTEGER KEYLENGTH:=8;
       INTEGER KEYLOCATION:21;
       INTEGER RELOP:=2;
       INTRINSIC FOPEN,FCLOSE,FREAD,FFINDBYKEY,READ,PRINT,
    FCHECK,FERRMSG,TERMINATE;
       <<************************>>
       <<* OPEN THE KSAM FILE *>>
       <<************************>>
       FILNUM:=FOPEN(FILNAME,3); <<OPEN THE KSAM FILE>>
       IF FILNUM=0
       THEN BEGIN <<CANNOT OPEN KSAM FILE>>
 MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
 PRINT(MESSAGE,-21,0);
 FCHECK(FILNUM,ERRORCODE),; <<GET THE ERROR NUMBER>>
 FEERMSG(ERRORCODE,MESSAGE,LENGTH); <<GET MESSAGE STRING>>
 PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
 TERMINATE;
     END;

<<*******************************************************>>
<<* READ DATA FROM KSAM FILE IN TELEPHONE # SEQUENCE      *>>
<<*******************************************************>>
       L2:
       FFINDBYKEY(FILNUM,KEYVALUE,KEYLOCATION,KEYLENGTH,RELOP);
       MOVE MESSAGE:="** LIST IN TELEPHONE NO. SEQUENCE";
       PRINT (MESSAGE,-33,0);
       L3:
       FREAD(FILNUM,INPUT,-72);  <<READ SEQUENTIALLY BY ALTERNATE KEY>>

       IF >
       THEN BEGIN         <<END OF FILE>>
```

Figure 4-2. FFINDBYKEY Example

```
FCLOSE(FILNUM,0,0);   <<CLOSE THE KSAM FILE>>
IF  < >  THEN
   BEGIN  <<CLOSE UNSUCCESSFUL>>
MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE";
PRINT(MESSAGE,-29,0);
FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
FERRMSG(ERRORCODE,MESSAGE,LENGTH) <<GET MESSAGE STRING>>
PRINT(MESSAGE,-LENGTH,0);  <<PRINT ERROR MESSAGE>>
  END;
 TERMINATE;
     END;
       IF  <
       THEN BEGIN
 MOVE MESSAGE;="ERROR OCCURRED WHILE READING INPUT";
 PRINT(MESSAGE,-34,0);
 TERMINATE;
     END;
       PRINT(OUTPUT,-72,0);
      <<**********************************>>
      <<*  GO BACK TO GET ANOTHER RECORD  *>>
      <<**********************************>>
       GO TO L3:
```

Figure 4-2. FFINDBYKEY Example (continued)

FFINDN

INTRINSIC NUMBER 301

Positions the logical record pointer to relative record number according to key sequence.

```
            IV      DV          IV
FFINDN(filenum,number,keylocation)
```

When FFINDN is executed, it positions the KSAM logical record pointer to the record whose relative record number is specified in the parameter *number*. Records are numbered from the record with the lowest key value in the key that starts at *keylocation* in each record. Record numbering starts with zero unless the flagword in the FOPEN *ksamparam* parameter specifies that record numbering starts with 1, or the FIRSTREC parameter in the >BUILD command is set to 1.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be positioned.

number *double by value (required)*

Relative record number counting from 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.

keylocation *integer by value (required)*

The relative byte location in the record of the key to be used. The first byte is byte 1. If keylocation is set to zero, the primary key is assumed.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FFINDN

When you specify the relative record number, it is important not to confuse this number with the chronological record number, the number of the record as it is stored in the file. To illustrate, assume a file in which records have been stored in chronological order from the beginning of the file (BOF). Each record has a key starting in byte 3 that contains a name. The relative record number is based on the value of this key, not the relative location of the record in the file.

For example:

```
FFINDN(FILNUM,4D,3)
```

This call positions the logical record pointer (as shown in Figure 4-3) to record number 4 of the key at location 3. Note that record number 4 is the fifth record in the sequence of key values:

ABLE	relative record 0D
BAKER	relative record 1D
CHARLIE	relative record 2D
DOG	relative record 3D
EASY	relative record 4D
FOX	relative record 5D

If you want to position the chronological record pointer to the relative record number in chronological sequence from the beginning of the file, you can use the intrinsic FPOINT, discussed later in this section. Chronological order is the order in which records are written. In Figure 4-3, record number 4 in key order, to which FFINDN positioned the file pointer, is also record number 2 in chronological order.

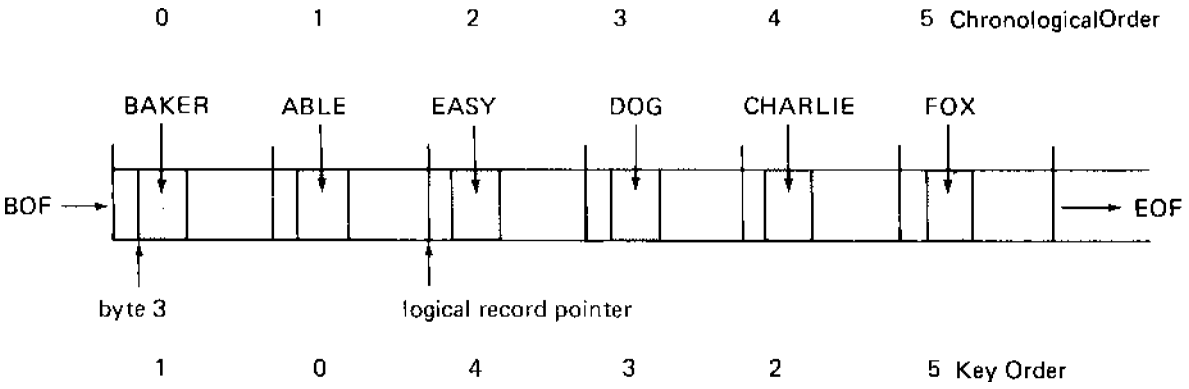


Figure 4-3. File Position with FFINDN

Note that FFINDN is useful to reset the pointer to an alternate key. For example, when you open the file, the primary key is selected by default. If you want to select another key starting in location 23 and position to the first record in key sequence, you can use the following command:

```
FFINDN(FILNUM,-1,23)
```

SHARED ACCESS

If you use FFINDN to position the pointer before calling another procedure that reads or updates the file in a shared environment, you must call FLOCK before calling FFINDN. Then, after performing the read or update operation, you should unlock the file so other users can access it. If you lock the file after calling FFINDN, another user can change the pointer position without your program being aware of it.

FGETINFO

INTRINSIC NUMBER 11

Requests access and status information about a file.

```
                IV      BA      L       L       I       I       L       L
FGETINFO (filenum,filename,foptions,aoptions,recsize,devtype,ldnum,hdaddr,

    I     D     D      D       D          D       I       L
filecode,recpt,eof,flimit,logcount,physcount,blksize,extsize,

     I          I          BA       D     0-V
numextents,userlabels,creatorid,labaddr) ;
```

Once a file is opened on any device, the FGETINFO intrinsic can be used to request access and status information about that file.

PARAMETERS

filenum — *integer by value (required)* A word identifier supplying the file number of the file about which information is requested.

filename — *byte array (optional)* A byte array to which is returned the actual designator of the file being referenced, in this format:

f.g.a

where

f = the local file name
g = the group name (supplied or implicit).
a = the account name (supplied or implicit).

The byte array must be 28 bytes long. When the actual designator is returned, unused bytes in the array are filled with blanks on the right.

Default: The actual designator is not returned.

foptions — *logical (optional)* The *foptions* parameter returns seven different file characteristics by setting corresponding bit groupings in a 16-bit word. Correspondence is from right to left. The file characteristics returned are the same as those specified for *foptions* in the FOPEN intrinsic (refer to Table 4-6, in the FOPEN description). Note that bit 4 is set to 1 to indicate a KSAM file.

Default: Foptions are not returned.

aoptions — *logical (optional)* The *aoptions* parameter returns up to seven different access options represented by bit groupings in a 16-bit word, as described for the *aoptions* parameter of FOPEN (refer to Table 4-7 in the FOPEN description).

Default: Aoptions are not returned.

recsize — *integer (optional)* A word to which is returned the logical record size associated with the file. If the file was created as a binary type, this value is

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

Default: The logical record size is not returned.

devtype *integer (optional)* A word to which is returned the type and subtype of device being used for the file, where

bits (0:8) = device subtype, and
bits (8:8) = device type.

If the file is not spooled, which can be determined from *hdaddr* (0:8), the returned *devtype* is actual. The same is true if the file is spooled and was opened via logical device number. However, if an output file is spooled and was opened by device class name, *devtype* contains the type and subtype of the first device in its class, which may be different from the device actually used.

Default: The device type and subtype are not returned.

ldnum *logical (optional)* A word to which is returned the logical device number associated with the device on which the file resides.

If the file is a disc file, then the logical device number will be that of the first extent. If the file is spooled, then *ldnum* will be a virtual device number which does not correspond to the system configuration I/O device list.

Default: The logical device number is not returned.

hdaddr *logical (optional)* A word to which the hardware address of the device is returned, where

bits (0:8) = the Device Reference Table (DRT) number, and
bits (8:8) = the unit number.

If the device is spooled, the DRT number will be zero and the unit number is undefined.

Default: The hardware address is not returned.

filecode *integer (optional)* A word to which is returned the value recorded with the file as its data file code (for disc files only).

Default: The file code is not returned.

recpt *double (optional)* A double word to which a double integer representing the current chronological record pointer setting is returned. Following a call to FREADC for example, *recpt* would be the chronological record number of the record just read by the FREADC intrinsic. Remember that chronological record numbers can begin with 0 or 1, depending on how the file was built.

Default: The chronological record pointer setting is not returned.

eof *double (optional)*

A double word to which is returned a double positive integer equal to the number of logical records currently in the data file. If the file does not reside on disc, this value is zero.

Default: The number of logical records in the file is not returned.

flimit *double (optional)* A double word to which is returned a double positive integer representing the number of the last logical record that could ever exist in the data file because of the physical limits of the file.

Default: The file limit information is not returned.

logcount *double (optional)* A double word to which is returned a double positive integer representing the total number of logical records passed to and from the user during the current access of the file.

Default: The logical record count is not returned.

physcount *double (optional)* A double word to which is returned a double positive integer representing the total number of physical input/output operations performed within this process against the file since the last FOPEN call.

Default: The number of I/O operations is not returned.

blksize *integer (optional)* A word to which is returned the block size associated with the file. If the file was created as a binary type, this value is positive and expresses the size in words. If the file was created as an ASCII type, this value is negative and shows the size in bytes.

Default: The block size is not returned.

extsize *logical (optional)* A word to which is returned the disc extent size associated with the data file (in sectors).

Default: The disc extent size is not returned.

numextent *integer (optional)* A word to which is returned the maximum number of disc extents allowable for the data file.

Default: The maximum allowable number of extents is not returned.

userlabels *integer (optional)* A word to which is returned the number of user header labels defined for the file when it was created. When an old file is opened for overwrite output, the value of *userlabels* is not reset and old user labels are not destroyed.

Default: The number of user labels is not returned.

creatorid *byte array (optional)* A type array to which is returned the eight-byte name of the user who created the file. If the file is not a disc file, blanks are returned.

Default: The user name is not returned.

labaddr *double (optional)* A double word to which is returned the sector address of the label of the file. The high-order eight bits show the logical device number. The remaining 24 bits show the absolute disc address.

Default: The label address is not returned.

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

USING FGETINFO

By calling FGETINFO you can return to your program any or all of the items listed as parameters. Except for the identifying *filenumber*, each of these parameters is optional. When omitted, embedded parameters are indicated by commas. Parameters omitted from the end of the list need not be so indicated. For example, to locate the number of records in the file by finding the end of file, you can call FGETINFO as follows:

```
FGETINFO(FILNUM , , , , , , , , , , LSTREC);
 \--------------  eof parameter
```

The value returned to LSTREC is the number of records in the file. The value LSTREC is also the chronological number of the last record in the file. This number can be used to read the last chronological record with FREADC or FREADDIR.

Another useful parameter of FGETINFO is *recpt*. This parameter returns the chronological record number of the record last read. The example in Figure 4-4 illustrates both these parameters. First, FGETINFO is used to determine the total number of records in the file using the parameter eof. Then, each record in the file is read in sequential order by primary key. Following each sequential read, FGETINFO retrieves the chronological record number of the record just read.

In the output from the program (refer to Figure 4-4), note that the record number returned by FGETINFO is the chronological number. For instance, the first record written to the file was the record with record number 1. This record, which contains the primary key value "NOLAN JACK", is the fourth consecutive record in key sequence.

```
$CONTROL MAIN=JEXAMPLS
<<*********************************************************>>
<<*                                                       *>>
<<*                        EXAMPLE 5                      *>>
<<*      FIND NUMBER OF RECORDS & RECORD NUMBER           *>>
<<*                                                       *>>
<<*********************************************************>>
INTEGER FILNUM:
INTEGER ERRORCODE,LENGTH;
BYTE ARRAY FILNAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
 DOUBLE EOP,RECPTR;
 INTRINSIC FOPEN,FCLOSE,FREAD,FGETINFO,DASCII;
 INTRINSIC PRINT,TERMINATE,FCHECK,FERRMSG;
<<************************>>
<<* OPEN THE KSAM FILE *>>
<<************************>>
FILNJM:=FOPEN(FILNAME,3); <<OPEN THE KSAM FILE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT (MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE): <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH0;<<CONVERT TO STRING>>
       PRINT (MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*************************************************>>
<<* FIND NUMBER OF RECORDS WRITTEN TO FILE *>>
<<*************************************************>>
 FGETINFO(FILNUM,,,,,,,,,EOF);
 IF  < >
 THEN BEGIN
        MOVE MESSAGE:="ERROR OCCURRED FINDING NUMBER
            OF RECORDS";
        PRINT(MESSAGE,-40,00;
        FCHECK(FILNUM,ERRORCODE);
        FERRMSG(ERRORCODE,MESSAGE,LENGTH);
        PRINT(MESSAGE,-LENGTH,0);
        TERMINATE;
      END;
```

Figure 4-4. FGETINFO Example

```
<<*************************************************>>
<<* PRINT NUMBER OF RECORDS IN FILE *>>
<<*************************************************>>
 MOVE MESSAGE:="RECORDS IN ENTITY = ";
 DASCII(EOF,10,MESSAGE(9));
 PRINT(MESSAGE,-20,0);
L1;
<<*************************************************>>
<<* READ KSAM SEQUENTIALLY *>>
<<*************************************************>>
FREAD(FILNUM,INPUT,-72);
IF >                    <<END OF DATA>>
THEN BEGIN
   FCLOSE(FILNUM,0,0);    <<CLOSE THE KSAM FILE>>
   IF <> THEN
     BEGIN
        MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE";
        PRINT(MESSAGE,-22,0);
        FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
        FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CINVERT TO STRING>>
        PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
     END;
   TERMINATE#
  END;
IF <
THEN BEGIN
       MOVE MESSAGE:="ERROR OCCURRED WHILE READING KSAM FILE":
       PRINT(MESSAGE,-37,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMRER>>
       FERRMSG(ERRORCODE,MESSAGE.LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE#-LENGTH,0): <<PRINTOUT ERRUR MESSAGE>>
       TERMINATE;
     END#
<<*************************************************>>
<<* WRITE THE DATA JUST READ FROM KSAM FILE *>>
<<*************************************************>>
PRINT(OUTPUT;-72,0):
```

Figure 4-4. FGETINFO Example (continued)

```
<<*********************************************************>>
<<* TO FIND OUT RECORD NUMBER OF THE RECORD JUST READ *>>
<<*********************************************************>>
 FGETINFO(FTLENUM,,,,,,,,RECPTR):
 IF  < >
 THEN BEGIN
           MOVE MESSAGE:="ERROR OCCURRED FINDING RECORD NUMBER":
           PRINT(MESSAGE,-36,0);
           FCHECK(FILNUM,ERRORCODE);
           FERRMGG(ERRORCODE,MESSAGE,LENGTH);
           PRINT(MESSAGE,-LENGTH,0);
           TERMINATE;
        END;
<<*********************************************>>
<<* PRINT THE RECORD NUMBER OF LAST RECORD READ *>>
<<*********************************************>>
 MOVE MESSAGE:="RECORD# = ";
 DASCII(RECPTR,10,MESSAGE(5) );
 PRINT(MESSAGE,-12,0);
<<*********************************>>
<<* GO BACK TO GET ANOTHER RECORD *>>
<<*********************************>>
GO TO L1;
!EOD;
!EOD
```

When Executed, the Following Output is Printed:

```
 RECORDS IN FILE= 10
 CARDIN RICK 578-7018 11100 WOLFF ROAD CUPERTINO CA. 94053
 RECORD# = 4
 ECKSTETN LEO 287-5137 5303 STEVENS CREEK SANTA CLARA CA. 95050
 RECORD# = 3
 HOSODA JOE 227-8214 1180 SAINT PETER CT. LOS ALTOS CA. 94022
 RECORD# = 2
 NOLAN JACK 923-4975 967 REED AVE. SUNNYVALE CA. 94087
 RECORD# = 1
 PASBY LINDA 295-1187 TOWN & CNTRY VILLAGEF SAN JOSE CA. 94102
 RECORD# = 5
 ROBERT GERRY 259-5535 12345 TELEGRAPH AVE BERKELEY CA. 90871
 RECORD# = 7
 SEELY HENRY 293-4220 1144 LEBERTY ST. EL CERRITO CA. 94053
 RECORD# = 6
 TURNEWR IVAN 984-8498 22905 EMERSON ST. OAKLAND CA. 98234
 RECORD# = R
 WESTEP ELDER 287-4598 1256 KINGFISHER ST. SUNNYVALE CA. 43098
 RECORD# = 10
 WHITE GORDON 398-0301 4350 ASHBY AVE. BERKELEY CA. 91234
 RECORD# = 9

   END OF PROGRAM
```

Figure 4-4. FGETINFO Example (continued)

FGETKEYINFO

INTRINSIC NUMBER 303

Requests access and status information about a KSAM file,

```
                          IV        A         A
FGETKEYINFO(filenum,ksamparam,ksamcontrol)
```

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the filenumber of the file about which information is requested.

ksamparam *array (required)*

An array of the same format and size as the byte array of the same name in FOPEN (refer to Table 4-8), except that key file size is given as the number of sectors. The length of the array depends on the number of keys in the KSAM file; its length is 17 words plus 4 words for each key. Note that the device (words 6-14) is not returned as a device class name but as an ASCII string containing the logical device number.

ksamcontrol *array (required)*

An array whose size is 128 words containing control information about the key file. Refer to Table 4-5 for the definition of the array contents.

CONDITION CODES

CCE	Request granted.
CCG	(not retuned)
CCL	Request denied because an error occurred such as: insufficient space declared for *ksamparam* or *ksamcontrol*; or an illegal file number; or the DB register is not set to the user stack.

USING FGETKEYINFO

Once a KSAM file is opened, you can request information about the key file through this intrinsic. The *ksamparam* return provides static information defined for the key file at the time it was created. The *ksamcontrol* parameter provides dynamic information about the use of the key file from the time it was created. In particular, it provides a count of the number of times the key file was referenced by various intrinsics, the date and time it was created, closed, updated or written to, and so forth.

Table 4-5. FGETKEYINFO *ksamcontrol* Parameter Format

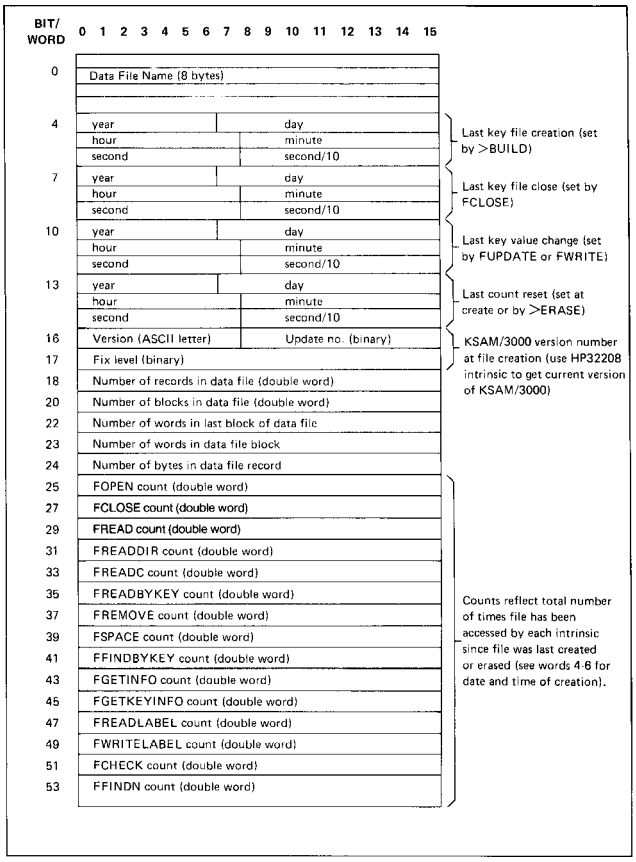
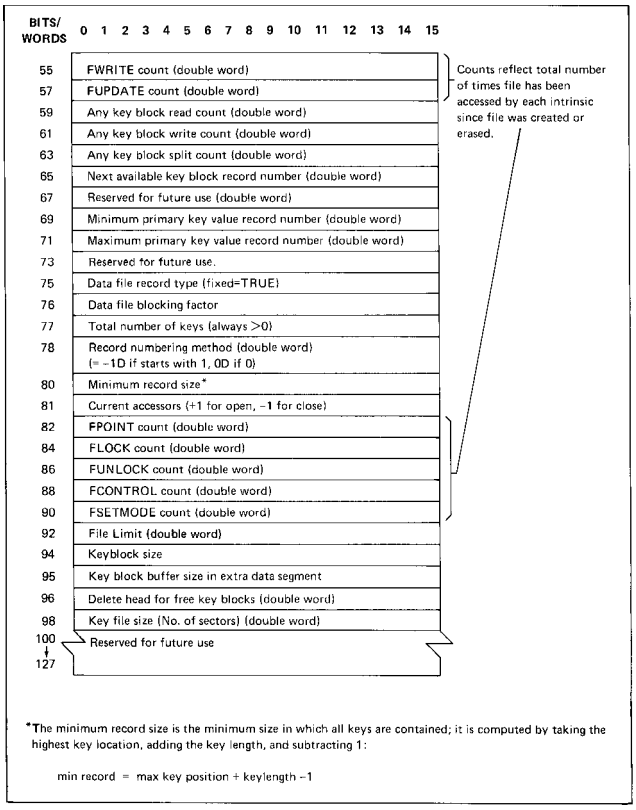


Table 4-5. FGETKEYINFO *ksamcontrol* Parameter Format (continued)



FLOCK

INTRINSIC NUMBER 15

Dynamically locks a file.

```
                IV        LV
FLOCK(filenum,lockcond);
```

The FLOCK intrinsic dynamically locks a file and transfers the latest control information from disc to the buffers. A call to FLOCK is required before any attempt is made to read or modify a file opened for shared access.

Warning **Always unlock a locked file (refer to FUNLOCK) after reading or modifying it. Not following this practice may result in corruption of the file.**

PARAMETERS

filenum *integer by value (required)*

A word supplying the file number of the file to be locked.

lockcond *logical by value (required)*

A word specifying conditional or unconditional locking:

TRUE— Locking will take place unconditionally. If the file cannot be locked immediately, the calling process suspends until the file can be locked.

Bit 15 = 1

FALSE— Locking will take place only if the file's Resource Identification Number (RIN) is not currently locked. If the RIN is locked, control returns immediately to the calling process, with condition code CCG.

Bit 15 = 0

CONDITION CODES

The condition codes possible when *lockcond* = TRUE are

CCE Request granted.

CCG Not returned when *lockcond* = TRUE.

CCL Request denied because this file was not opened with the *dynamic locking aoption* specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN Capability.

The condition codes possible if *lockcond* = FALSE are

CCE Request granted.

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

CCL	Request denied because: this file was not opened with the *dynamic locking aoption* specified in the FOPEN intrinsic; or the request was to lock more than one file and the calling process does not possess the Multiple RIN Capability.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

Standard Capability sufficient if only one file is to be locked dynamically.

If more than one file is to be locked dynamically, the Multiple RIN Capability is required.

USING FLOCK

The dynamic locking and unlocking capability allows you to complete any update to a file when it is possible for other users to access the file. When dynamic locking is allowed (bit 10 of FOPEN *aoptions* parameter is set to allow dynamic locking); then you must use the FLOCK intrinsic to lock the file before writing, rewriting, or deleting any records. This requirement insures that another user does not attempt to change the same record at the same time. FLOCK also insures that the most recent data is available in the file. A locked file can be unlocked following the update with the FUNLOCK intrinsic.

When FLOCK is executed, it clears all the buffers and transfers the latest control information from the KSAM file to the buffers. This insures that any subsequent read of the file retrieves the latest information from disc rather than from the buffers. (Note that FCONTROL control code 7 also clears the buffers.)

If you use the Multiple RIN capability, a sequence of file locking should be agreed upon or you should use conditional locking (*lockcond* = FALSE). Otherwise, it is possible to lock other users from the file. Consider the situation where one program unconditionally locks file A and then attempts to lock file B. If another program unconditionally locks file B and then attempts to lock file A, both programs will wait indefinitely to lock the second file in sequence. To avoid this, both programs should agree to lock the files in the sequence A first, then B; or both programs should use only conditional locks.

For example, suppose you open a KSAM file called DATA1 for shared access in update mode and allow dynamic locking and unlocking:

```
FIL1:=FOPEN(DATA1,7,%345);
```

The parameters specified are:

filenum	File number of DATA1, which is assigned to FIL1 when the file is opened.
formal-designator	Name identifying the file contained in DATA1.
foptions	The value 7 specifies that this is an old user file (bits 14,15 = 11) and that it is coded in ASCII (bit 13 = 1).
aoptions	The octal value 345 indicates that the file was opened for update (bits 12 through 15 = 0101), that dynamic locking/unlocking is allowed (bit 10 = 1), and that access is shared (bits 8 and 9 = 11).

This file can then be locked as follows:

```
FLOCK(FIL1,1)
```

The parameters specified are:

filenum	File number of file DATA1 contained in the variable FIL1.
lockcond	= 1 which means the file is to be locked unconditionally. If the file cannot be locked immediately, the calling process is suspended until the file can be locked.

FOPEN

INTRINSIC NUMBER 1

Opens a file.

```
    I                   BA              LV      LV        IV       BA        BA
filenum:=FOPEN(formaldesignator,foptions,aoptions,recsize,device,ksamparam,

    IV          IV          IV         DV        IV
userlabels,blockfactor,numbuffers,filesize,numextents,

    IV         IV      O-V
initialloc,filecode);
```

The FOPEN intrinsic makes it possible to access a KSAM file. In the FOPEN intrinsic call, a particular file may be referenced by its *formal file designator*. When the FOPEN intrinsic is executed, it returns to the user's process a *file number* by which the system uniquely identifies the file. This file number, rather than the file designator, then is used by subsequent intrinsics in referencing the file.

FUNCTIONAL RETURN

This intrinsic returns an integer file number used to identify the opened file in other intrinsic calls.

PARAMETERS

formal-designator — *byte array (required)*

Contains a string of ASCII characters interpreted as a formal file designator. This string must begin with a letter, contain alphanumeric characters, slashes, or periods, and terminate with any nonnumeric character except a slash or a period. If the string names a userpredefined file, it can begin with an asterisk (*). Note: The DEL, SAVE, or TEMP parameters should not be used to predefine a KSAM file in a :FILE command; they will cause deletion or duplication of the file.

foptions — *logical by value (optional)*

The *foptions* parameter allows you to specify different file characteristics, by setting corresponding bit groupings in a 16-bit word. If the file is new, bit 4 must be set to 1 to indicate that this is a KSAM file. Refer to Table 4-6 for the *foption* bit settings.

Default: All bits are set to zero.

aoptions — *logical by value (optional)*

The *aoptions* parameter permits you to specify the various access options established by bit groupings in a 16-bit word. These access options are defined in Table 4-7.

Default: All bits are set to zero.

recsize — *integer by value (optional)*

An integer indicating the size of the logical records in the data file. If a positive number, this represents words; bytes are represented by a negative number. If the file is a newly-created file, this value is recorded permanently in the file label. If the records in the file are of variable length, this value indicates the maximum logical record length allowed.

Binary files are word oriented. A record size specifying an odd byte count for a binary file is rounded up by FOPEN to the next highest even number.

ASCII files may be created with logical records which are an odd number of bytes in length. Within each block, however, records begin on word boundaries.

For either ASCII or binary files with fixed-length records, the record size is rounded up to the nearest word boundary. For example, a *recsize* specified as −106 for an ASCII file is 106 characters (53 words) in length. A *recsize* of −113 for a binary file is 114 characters (57 words) in length. The rounded sizes should be used in computations for *blockfactor* or block size.

Default: The default value is the configured physical record width of the associated device.

device *byte array (optional)*

Contains a string of ASCII characters terminated by any nonalphanumeric character (except a slash or period) that designates the device on which the file is to reside. It may be a device class name of up to eight alphanumeric characters beginning with a letter; or a logical device number consisting of a three-byte numeric string; or a remote device identifier consisting of the device class name or logical device number followed by a pound sign (#) and a remote device class name or logical device number.

Device class names and logical device numbers are assigned to devices during system configuration.

For KSAM files, the device must be a random access device such as the disc. If the file is a newly-created disc file specified as a device class name, then all extents to the file must be members of the same class. Similarly, if the device is identified by logical device number, then all extents must have the same logical device number.

Default: Disc.

ksamparam *byte array (optional)*

Contains information describing the key file of a KSAM file. It includes the key file name, size and device plus an entry for the primary key and up to 15 alternate keys. If the file is new (is being created by FOPEN) then this array must be included. If the file is an old file, it can be omitted. Note that if the parameter is included and the file is not a KSAM file, an error can result. Refer to Table 4-8 for a full description of *ksamparam*.

Default: key file description is omitted.

userlabels *integer by ualue (optional)*

Specifies the number of user-label records to be written for the data file. If there are no user labels, this parameter can be omitted.

Default: The default number of user-label records is zero.

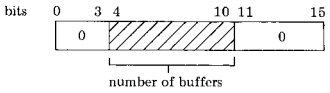
blockfactor *integer by value (optional)*

Establishes the size of each block in the data file by specifying the number of logical records per block. It also establishes the size of the data file buffer in KSAM's extra data segment . For fixed-length records, *blockfactor* is the actual number of records in a block; for variablelength records, *blockfactor* is a multiplier used to compute block size from record size; ((maximum *recsize* +1) * *blockfactor*) +1 = blocksize. The value of *blockfactor* should be an integer that results in a block size less than 4K words. The blockfactor is from 1 through 255. If you specify a negative value or zero, the default value is used. Values greater than 255 are defaulted to the specified blockfactor modulo 256.

Default: 1

numbuffers *integer by value (optional)*

An integer between 1 and 20 that specifies the number of key block buffers in the extra data segment used by KSAM files for buffering data and key blocks. The number of buffers is specified in bits 4-10; the rest of the word must be set to zeros:



This number should only be specified if the default number assigned by KSAM affects performance. Refer to appendix B, under KSAM Extra Data Segments for a discussion of how the key block buffers are used.

Default: Between 1 and 20 bUffers depending on access type, number of keys, and number of levels per hey. (Refer to appendix B.)

filesize *double by value (optional)*

A double-word integer specifying the maximum data file size as the number of logical records in the file. A zero or negative value results in the default *filesize* setting. The maximum file capacity is over two million (2^{21}) sectors; a sector contains 128 words.

Default: 1024 logical records

numextents *integer by value (optional)*

An integer specifying the number of extents (integral number of contiguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The number of extents applies equally to the data and key files on the assumption that there is a proportional expansion in each. The size of each extent is determined by the *filesize* parameter value divided by the *numextents* parameter value. If specified,

numextents must be an integer from 1 to 32. A zero or negative value results in the default setting.

Default: 8 extents.

Note Extents are allocated on any disc in the device class specified in the *device* parameter when the file was created. If it is necessary to insure that all extents of a file are on a particular disc, a single disc device class or a logical device number must be used in the *device* parameter.

initialloc *integer by value (optional)*

An integer specifying the number of extents to be allocated to the data file when it is opened. (For a key file, this parameter is forced equal to the value of *numextents*.) This must be an integer from 1 to 32. If an attempt to allocate the requested disc space fails, the FOPEN intrinsic returns an error condition code to the calling program.

Default: 1 extent.

filecode *integer by value (optional)*

An integer recorded in the file label and made available for general use to anyone accessing the file through the FGETINFO intrinsic. This parameter is used for new data files only. The filecode applies to data files only; the key file code is always 1080 and need not be specified. For this parameter, any user can specify a non-negative integer.

Default: 0

CONDITION CODES

CCE Request granted. The file is open.

CCG Not returned by this intrinsic.

CCL Request denied. This may be because another process already has exclusive or semi-exclusive access for this file, or an initial allocation of disc space cannot be made due to lack of disc space. The file number value returned by FOPEN if the file is not opened successfully is zero. The FCHECK intrinsic should be called for more details.

USING FOPEN

FOPTIONS PARAMETER

Table 4-6. FOPEN *foptions* Parameter Format

BITS 0 1 2 3	4	5	6	7	8 9	10 11 12	13	14 15
	KSAM FILE	DIS-ALLOW :FILE		CCTL	RECORD FORMAT	DEFAULT FILE DESIGNATOR	ASCII/ BINARY	DOMAIN

Set shaded areas to zero for KSAM files.

BITS	OPTION	SETTINGS
14:2	File Domain	00 = New file created by FOPEN call. No search is required. The *ksamparam* parameter must be present to define the file structure. (*default*) 01 = Old permanent file; search system file domain. 10 = Old temporary file; search job file domain. 11 = Old user file; search job file domain and, if not found, search system file domain.
13:1	ASCII/Binary	0 = Binary code used to record data; any dummy records are padded with zeros. (*default*) 1 = ASCII code used to record data; any dummy records are padded with blanks.
10:3	Default File Designator	000 = Actual file designator is the same as the formal file designator. (*default and only setting allowed for KSAM files.*)
8:2	Record Format	00 = Fixed-length records. (*default*) 01 = Variable-length records. (Other settings not allowed for KSAM files)
7:1	CCTL	0 = Carriage control directive not expected. (*default and only setting allowed for KSAM files.*)
5:1	Disallow :FILE	0 = Allow :FILE command to override FOPEN file specifications. Note that *formaldesignator* is the only :FILE specification allowed for KSAM files. (*default*) 1 = Disallow (ignore) :FILE command file equations when in conflict with FOPEN.
4:1	KSAM	0 = Not a new KSAM file. (*default*) 1 = New KSAM file or exising KSAM file opened as an MPE file.

Table 4-7. FOPEN *aoptions* Parameter Format

BITS 0 1 2	3	4	5 6 7	8 9	10	11	12 13 14 15
	KSAM ACCESS	NO WAIT		EXCLUSIVE ACCESS	DYN-AMIC LOCK		ACCESS TYPE

Set Shaded areas to zero for KSAM files.

BITS	OPTION	SETTINGS
12:4	Access Type	0000 = Read only. (*default*). Allows access to all intrinsics except: FWRITE,FUPDATE, and FREMOVE. 0001 = Write only. Delete previously written data. Allows access to all intrinsics except: FREAD,FREADDIR,FREADC,FREADBYKEY, FUPDATE,FREMOVE,FSPACE,FPOINT,FFINDBYKEY, and FFINDN. 0010 = Write only. Save previously written data. Allows access to same intrinsics as write only with delete. 0011 = Same as above. 0100 = Input/Output access. Allows access to all intrinsics except: FUPDATE and FREMOVE. 0101 = Update access. Allows access to all intrinsics.
10:1	Dynamic Locking	0 = Disallow dynamic locking/unlocking. (*default*) *1 = Allow dynamic locking/unlocking. Allows use of FLOCK and FUNLOCK intrinsics to permit or restrict concurrent access to file.
8:2	Exclusive Access	00 = Default access depending on access type: if access type = 0000 (read only) default is 11 (share access); if access type is any other, default is 01 (exclusive access). 01 = Exclusive access. Prohibits another FOPEN request to open the file until current process issues FCLOSE or terminates. 10 = Semi-exclusive access. Allows another process to open this file for read only but prohibits any output access until this process issues FCLOSE or terminates. *11 = Share access. After file is opened, permits concurrent access to the file by any process in any access mode, subject only to MPE security provisions in effect.

Table 4-7. FOPEN *aoptions* Parameter Format (continued)

BITS	OPTION	SETTINGS
4:1	No Wait	0 = No Wait input/output. (*default and only setting allowed for KSAM files.*)
3:1	KSAM Access	0 = KSAM access expected. 1 = Non-KSAM access expected; KSAM key file or data file is treated as standard MPE file. For this setting to be meaningful, file must be a KSAM file (*foptions* 4:1 = 1).

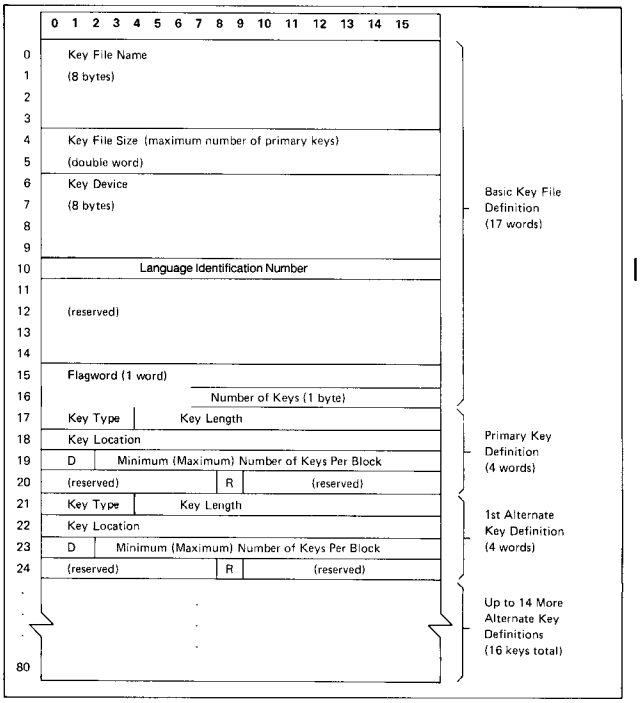
* If dynamic locking is enabled with share access, a call to FLOCK must precede any call to FREMOVE, FUPDATE, or FWRITE. Note that a file equation that specifies shared access (FILE filename;SHR), automatically sets the dynamic locking option, forcing users to lock for all access. Also, if you specify SHR (aoptions.(8:2)), KSAM will automatically set lock bit (aoptions.(10:1)) which will require that the file be locked before issuing any intrinsics.

KEY FILE DEFINITION

The *ksamparam* array defines the key file for a new KSAM file. If the file has already been created, this parameter can be set to all zeros or omitted. Otherwise, it must be assigned values to define the key file as shown in Table 4-8.

When a new KSAM file is created, the MPE end-of-file for the key file is set to the file limit. The file limit is based on the key file size (see words 4-5 of *ksamparam*). The location of the key file end-of-file can be determined by executing the VERIFY command of KSAMUTIL and looking at the heading KEY FILE EOF. A call to FGETKEYINFO returns the key file size as the number of sectors used by the file.

Table 4-8. FOPEN *ksamparam* Parameter Format



This array defines the key file portion of a new KSAM file being created by the FOPEN call. The values are:

Key File Name 8-byte file name that must be present if this is a new file. Only the name is specified; the account, group, and security are taken from the data file formal file designator.

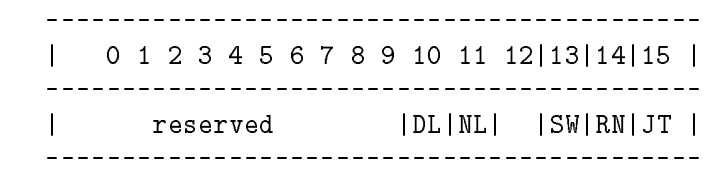
Key File Size — Double-word specifying the maximum number of primary keys expected from which the key file size is derived. If zero, the data file size is used.

(Note that a call to FGETKEYINFO returns the key file size as the number of sectors in the file.)

Key Device — 8-byte array that specifies the device on which the key file resides. Specified as a device class name of 1-8 alphanumeric characters beginning with a letter and terminated by a non-alphanumeric character such as a blank; or it is specified as a logical device number (3-byte numeric string) identifying a particular device. If the data file is assigned to a remote device the key file is automatically allocated to the same machine. *Default is DISC.*

Language Identification-Number — The number of the language to use in determining the key sequence. Bit 11 in the flagword must be set to 1 to use any language other than NATIVE-3000.If bit 11 is set to 0, NATIVE-3000 (the default) is used and the language identification number in word 10 is ignored. If the language is not configured, FOPEN returns condition code CCL. For detailed information on using Native Language Support (NLS) with KSAM files, refer to Appendix F.

Flagword — 1-word that specifies file characteristics as shown below:



JT — bit 15:1 = 1 if file is job temporary file

= 0 if file is a permanent file (*default*)

RN — 14:1 = 1 if record numbering starts with 1.

0 if record numbering starts with 0. (*default*)

SW — 13:1 = 1 if only sequential writing by primary key value is allowed.

= 0 if random writing by primary key value is allowed. (*default*)

0:9 = 0 all reserved bits must be set to 0.

NL — bit 11:1 = 1 if a configured language is specified.

= 0 if the default language (NATIVE-3000) is to be used.

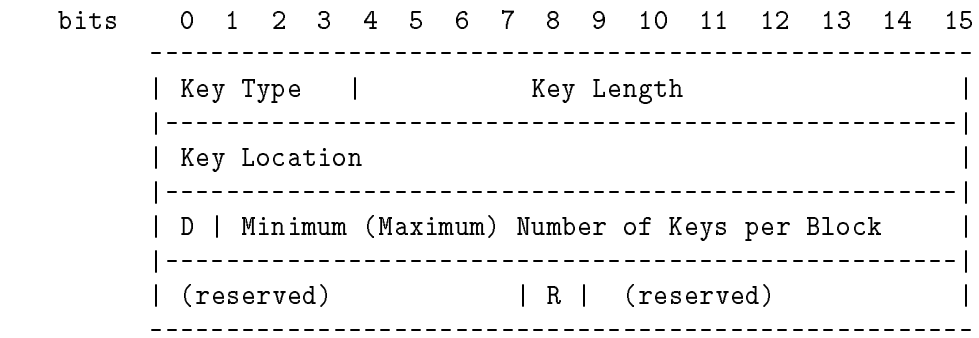
DL — bit 10:1 = 1 Duplicate file locks from different open paths of the same PIN are not allowed.

= 0 Duplicate file locks from the same PIN are always allowed.

Number of Keys — 1 byte providing the total number of keys for the file, specified as a numeric digit between 1 and 16, (left byte of word must be zero).

Key Definitions — Each key in the file requires a 4-word definition. The first definition is always of the primary key. Subsequent definitions describe any alternate keys. Up to

15 alternate keys are allowed in any one key file. The key definitions each contain the following information:



The information for each key has the form shown above starting in word 17 of *ksamparam*. It is defined as follows:

Key Type — 4 bits specifying the type of the key by the following code:

bits 0:4 =

0001 (1) = Byte key (1 to 255 bytes)

0010 (2) = Integer key (2 bytes)

0011 (3) = Double Integer key (4 bytes)

0100 (4) = Real key (4 bytes)

0101 (5) = Long key (8 bytes)

0110 (6) = Numeric Display key (1 to 28 bytes)

0111 (7) = Packed Decimal key, odd number of digits (1 to 14 bytes)

1000 (8) = Packed Decimal key, even number of digits (2 to 14 bytes)

Refer to Table 2-2 in section II for a full description of key type.

Key Length — 12 bits specifying length of the key in bytes. Length is a function of key type (see key type) but must never exceed 255 bytes.

Key Location — 1 word specifying the location of the first byte of the key in the record. Bytes in a record are numbered starting with 1. (Note that it is good practice to leave the first two bytes of a record empty of keys since these bytes are used by FREMOVE for the record delete code.)

D(Duplicate Flag) — 1 bit that determines if duplicate values are allowed for this key: = 0 if duplicate key values are not allowed (*default*) = 1 if duplicate key values are allowed.

Minimum (Maximum)-Number of Keys per-Block — 15 bits that specify the minimum number of keys allowed per key block. The value must be an even -numbered integer greater than or equal to 4. The resulting key block size must be less than 2048 words. If the resulting key block size is greater than 2048 words, the system will print the error message "INVALID KEY BLOCKING FACTOR VALUE (FSERR 190)" and will not open the file. In order to make optimum use of disc space, KSAM may increase the value specified here. If KSAM increases the number of keys per block, this new value is the *maximum size* of the key block. (Refer to

appendix B for particulars on the calculation of block size and the adjustment of the blocking factor). *The default generates a block size of 1K (1024) words.*

R(Random Insert Flag) 1 bit (8:1) that determines whether duplicate key is to be inserted randomly in duplicate key chain or is to be added to the end of the chain; the duplicate flag (D bit) must be set to 1 in order to use this flag.

= 0 if duplicate key values are to be inserted at the end of the chain (*default*)
= 1 if duplicate key values are to be inserted randomly. If inserted randomly, the chronological order of duplicate keys is no longer maintained, but the addition of keys is faster.

OPENING A NEW FILE

When FOPEN is used to open a new KSAM file, you must provide all the information needed to create the two files that make up a KSAM file: the key file and the data file. To inform the system that this is a KSAM file, the KSAM bit must be set in the *foptions* parameter; and the *ksamparam* parameter must be included to define the key file.

Figure 4-5 is a short SPL program that builds a KSAM file. The file has two keys; the primary key starting in column 1 is 20 characters long, and the alternate key starting in column 21 is 8 characters long. The primary key will contain a name, the alternate a phone number.

The first step is to declare all arrays and variables needed by the program followed by the intrinsic declaration for FOPEN. The shaded declarations in Figure 4-5 show these required to open the file; others are used in parts of the program not shown in this figure.

The next step is to move the necessary values to *ksamparam* in order to define the key file.

The last step is to call the FOPEN intrinsic, passing any previously defined variables or arrays by reference and passing all others by value.

DECLARATIONS FOR FOPEN

The array *ksamparam* is defined three different ways: as a numeric array containing 25 words (KSAMPARAMA), as a byte array equivalenced to the numeric array (KSAMPARAM), and as a double array also equivalenced to the numeric array (KSAMPARAMD). These three definitions allow the array to be addressed by word, by byte, or by double word as required.

The variable to which the file number is returned is declared to be an integer.

The two arrays that will contain the *formal designator* and *device* parameter values are declared and assigned these values. In this case, the formal designator is assigned the value JEXAMFIL. This name identifies both the KSAM file in its entirety and the data file if referenced separately. The device class name assigned to the *device* parameter is DISC.

Finally, the intrinsic itself is declared in an INTRINSIC statement.

DEFINING KSAMPARAM

The *ksamparam* parameter is assigned a variety of values that, for the sake of clarity, are assigned in separate statements. The values assigned to *ksamparam* define the key file. The statements that move values to *ksamparam* (refer to Figure 4-5) tell the system everything it needs to know in order to build the key file.

The first item moved to *ksamparam* is the key file name, up to 8 characters enclosed in quotes. In this case, the key file name is JKEYFILE.

Next, the size of the key file is defined in terms of the maximum number of primary keys expected. The size is specified as a double word integer and is assigned to the third double word in the array, specified by an index of 2 counting from double word 0. The maximum number of primary keys should be the same as the maximum number of records specified in the *filesize* parameter of FOPEN. KSAM assigns a key file size based on this value. If there are alternate keys, the key file size is made proportionately larger. If the key file size is specified as zero, KSAM uses the value of the FOPEN *filesize* parameter as the key file size.

The device class name is assigned in the 8 bytes starting in byte 12 that are allocated to device description. In this case, the device class name is DISC, the same as the device class name specified in the *device* parameter of FOPEN for the data file.

```
$CONTROL MAIN-JEXAMPL1
<<*********************************************************>>
<<*                                                       *>>
<<*              EXAMPLE 1                                *>>
<<*            BUILD A KSAM FILE                          *>>
<<*                                                       *>>
<<*********************************************************>>
ARRAY KSAMPARAMA(0:25);
BYTE ARRAY KSAMPARAM (*)=KSAMPARAMA;
DOUBLE ARRAY KSAMPARAMD(*)=KSAMPARAMA;
INTEGER FILNUM;
INTEGER ERRORCODE;
INTEGER LENGTH;
BYTE ARRAY FILENAME(0:9):="JEXAMFIL ";
BYTE ARRAY DEVICE(0:9):="DISC ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
INTRINSIC FOPEN,FCLOSE,PWRITE,READ,PRINT,TERMINATE;
INTRINSIC FCHECK,FERRMSG;
<<*******************************>>
<<* SETUP KSAMPARAM FOR FOPEN   *>>
<<*******************************>>
MOVE KSAMPARAM;="JKEYFILE";           <<THE KEY FILE NAME>>
KSAMPARAMD(2):=100D;                  <<THE MAX. # OF PRIMARY KEYS>>
MOVE KSAMPARAM(12):="DISC";           <<THE KEY FILE DEVICE TYPE>>
KSAMPARAMA(15):=&%(2)000000000000010; <<THE FLAG WORD>>
KSAMPARAMA(16):=2;                    <<PRIMARY KEY & ONE ALTERNATE>>
MOVE KSAMPARAMA(17):=([4/1,12/20],   <<TYPE=ASCII; LENGTH=20 BYTES>>
                       1,          <<KEY LOCATION START FROM COL 1>>
                      [1/0,15/4],  <<DUP NOT ALLOW; 4 KEY/BLOCK>>
                       0);         <<RESERVED>>
MOVE KSAMPARAMA(21):=([4/1,12/8],  <<TYPE=ASCII; LENGTH=20 BYTES>>
                       21,         <<KEY LOCATION START FROM COL 21>>
                      [1/0,15/4],  <<DUP NOT ALLOW; 4 KEY/BLOCK>>
                       0);         <<RESERVED>>
<<************************>>
<<* OPEN THE KSAM FILE *>>
<<************************>>
  FILNJM:=FOPEN(FILENAME,         <<THE DATA FILE NAME>>
          &%(2)0000100000000100,  <<KSAM,ASCII,NEW FILE>>
          &%(2)0000000001000001,  <<KSAM ACCESS,EXCLUSIVE,WRITE>>
          -72,              <<RECORD 72 BYTES LONG>>
          DEVICE,                 <<DEVICE=DISC>>
          KSAMPARAM,              <<THIS DESCRIBES THE KEYS>>
```

Figure 4-5. FOPEN Example—; Building a KSAM file

```
        ,
NO USERLABELS
        10,
BLOCK 10 RECORDS
        ,
NUMBUFFERS NOT USED
        100D);
THIS FILE CAN HOLD 100 RECORD
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH); <<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
```

Figure 4-5. FOPEN Example—; Building a KSAM file (continued)

Word 15, the flag word, is set next, It uses bits 13,14, and 15 to define three conditions of the key file. In this example, bit 14 is the only bit set. This means that record numbers in the file start with 1 rather than 0 (bit 14=1), that the file is a permanent file saved in the system directory (bit 15=0), and that records may be written to the file in random order rather than being restricted to ascending sequence by primary key (bit 13=0).In Figure 4-5 the flag word is specified as a binary value for clarity; it could have been specified as octal 2 (2 or %2) for brevity.

The right byte of the 16th word (byte 33) is set to 2 to specify that two keys are to be used: the primary key and one alternate.

This completes the general description of the file. Its name, size, device type, special conditions, and number of keys are now specified. The remainder of *ksamparam* defines each key in 4-word entries. The first entry always describes the primary key. Subsequent entries define up to 15 alternate keys. In this case, one primary and one alternate are defined.

Starting in word 17, the primary key is defined as type ASCII, 20 bytes long, its location starting in the first character of each record, and duplicate values are not allowed. It is blocked with four keys per block.

Starting in word 21, the altemate key is defined as type ASCII, 8 bytes long, located starting in character 21 of the record, duplicate values not allowed, and blocked four keys per block.

Refer to Table 4-8 for an illustration of the bit patterns used to define the *ksamparam* entries.

CALLING FOPEN

When all the variables and arrays that pass values by reference have been defined, the intrinsic FOPEN can be called. In Figure 4-5, each parameter is shown on a separate line and documented for clarity, but the call could also be specified as:

```
FILNUM:=FOPEN(FILNAME,%4004,%101,--72,DEVICE,KSAMPARAM,,10,0,100D);
```

This call is identical to the call in Figure 4-5 except that octal values are used for *foption* and *aoption.*

foptions

The value of *foptions* is set to octal 4004, for which the bit pattern is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	Binary
0		0			4			0			0			4		Octal

This specification defines the following file options:

New KSAM file (bit 4=1)
Allow: FILE (bit 5=0)
Fixed-Length Records (bits 8,9=00)
ASCII code (bit 13=1)
New file (bits 14,15=00)

aoptions

The value of *aoptions* is set to octal 101, for which the bit pattern is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	Binary
0		0			0			1			0			1		Octal

This specification defines the following access options:

KSAM access expected (bit 3=0)
Exclusive access (bits 8,9=01)
Dynamic locking not allowed (bit 10=0)
Access type is write only (bits 12-15=0001)

OPENING AN EXISTING FILE

Once the file has been created, opening it again after it has been closed is a simple process. The record size, device, blocking, buffersize, and file size are all defined for the data file. Therefore, these parameters need not be repeated. The key file has already been defined so that *ksamparam* need not be specified. This leaves the first three parameters to specify. Of these, only the *formal-designator* and the domain and KSAM options of the *foptions* parameter are always required. The *formal-designator* provides the file name in order to identify the file. The domain option specifies where to locate the file; if domain is set to zeros, the system expects a new file. Set the *foptions* KSAM option (bit 4:1) to 0 to indicate that

the file is not new If the file is to be read only, the access mode parameter, *aoptions*, can be omitted. For any other type of access, *aoptions* should be specified.

OPENING FILE FOR READ ACCESS

The example in Figure 4-6 illustrates opening a file for readonly access.

```
%CONTROL MAIN=JEXAMPL2                                              OPEN THE KSAM FILE
<<************************************************************************>>
<<*                                                                      *>>
<<*                                                                      *>>
<<************************************************************************>>
  INTEGER FILNUM;
  INTEGER ERRORCODE,LENGTH;
  BYTE ARRAY FILNAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
BYTE ARRAY KEYVALUE(0:7):="000-0000";
INTEGER KEYLENGTH:=8;
INTEGER KEYLOCATION:=21;
INTEGER RELOP:=2;
INTRINSIC FOPEN,FCLOSE,FREAD,FFINDBYKEY,READ,PRINT,
          FCHECK,FERRMSG,PRINT'FILE'INFO,TERMINATE;
<<*************************>>
<<* OPEN THE KSAM FILE   *>>
<<*************************>>
  FFILNUM:=OPEN(FILNAME,3);
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH); <<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
```

Figure 4-6. FOPEN Example—; Opening an Existing File

The file name is specified in the FILNAME array declaration as JEXAMFIL. This is the file that was created and opened for write-only access in Figure 4-5. It is opened for read-only access with the call:

```
FILNUM:=FOPEN(FILNAME,3);
```

The value of *foptions* is set to the value 3, for which the bit pattern is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	Binary
0	0			0			0			0			3			Octal

This specification defines the following file options:

Not a new KSAM file (bit 4=0)
Old user file (bits 14-15=11)

Because this is an existing (old) user file, other *foptions* settings defined when the file was created need not be respecified. For example, at creation the file was defined as containing ASCII code (bit 13=1). In subsequent FOPEN calls this bit can be 0 without changing the code to binary.

When an old user file is opened, the job file domain is searched first and then the system file domain is searched for the file specified in the formal designator.

The access parameter, *aoptions*, is not specified, but by default it specifies the following access mode:

KSAM access expected
Share access (default for read-only)
Read-only access

OPENING FILE FOR WRITE ACCESS

To open an existing file for write access, you use the same *foptions* values as you do to open the file for read-only access. The different access mode is specified in the *aoptions* parameter.

For example, assuming FILNUM and FILNAME have been declared:

```
FILNUM:=FOPEN(FILNAME,3,1)
```

The *foptions* specification is the same as described above. The *aoptions* specification is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Binary
0	0			0			0			0			1			Octal

This bit pattem defines the following access options:

KSAM access expected (bit 3=0)
Exclusive access (default for all access modes except read-only) (bits 8-9=00)
Disallow dynamic locking (bit 10=0)
Write only access (bits 12-15=0001)

This opens the file for write-only access in which all previous data is deleted. It is the access mode to use when writing to a file for the first time. If you want to write to the end of an existing file then bits 12-15 should equal 0010 and *aoptions* could be specified as 2 if other *aoptions* values are defaulted. To open the file for both reading and writing, bits 12-15 should be set to 0100, or the value 4. For update, these bits are set to 0101, or the value 5.

OPENING KSAM FILE AS MPE FILE

You may want to open either the key file or the data file as a standard MPE file. To do this, name the file you want to open in the *formaldesignator* parameter, set *foptions* bit 4:1 to 1, and then set *aoptions* bit 3:1 to 1. These settings indicate that the file is a KSAM file, but is to be treated as an MPE file. The remaining parameter settings depend on what you want to do with the open file. For example, if you want to read the key file, JKEYFILE, as an MPE file, you call FOPEN as follows:

```
INTEGER          FILNUM;
BYTE ARRAY       FILNAME(0:9):="JKEYFILE ";
:
INTRINSIC FOPEN,...;
:
```

```
FILNUM:=FOPEN(FILNAME,%4003,%10000);
```

The value of *foptions* defines the following file options:

Specified as KSAM file (bit 4=1)
Old user file (bits 14-15=11)

The value of *aoptions* indicates the following:

Non-KSAM access expected (bit 3=1)
Share access (default for read only bits 8-9=00)
Read-only access (default bits 12-15=0000)

Normally, the only time you need to set bit 4 of *foptions* to 1 is when you are originally creating a KSAM file. However, when you are opening an existing KSAM file for non-KSAM access, you must set this bit to 1 so that the system can distinguish the KSAM data or key file from an MPE file.

OPENING FILE FOR SHARED ACCESS

When a file is opened for shared access (*aoptions* bits 8,9=11), and you plan to modify the file in any way, you must enable dynamic locking (*aoptions* bit 10=1). This is necessary since you cannot call FWRITE, FUPDATE, or FREMOVE to modify a shared file without first calling FLOCK to lock the file.

Even if you are not planning to modify the file, but only plan to read it sequentially, you should allow dynamic locking when you open the file. This is because FREAD (as well as FUPDATE and FREMOVE) is a *pointer-dependent* procedure. Any time you call a pointer-dependent procedure (refer to Table 4-2), you must precede it with a call to a pointer-independent procedure that positions the pointer. It is important to call FLOCK to lock the file before setting the pointer with the pointer-independent procedure and leave it locked until you have completed the sequential read or update. This insures that no other user changes the position of the pointer between the call that positions the pointer and the call that depends on the pointer.

FPOINT

INTRINSIC NUMBER 6

Sets the chronological (and logical) record pointer for a KSAM file.

```
                    IV      DV
FPOINT(filenum,recnum);
```

The FPOINT intrinsic sets the chronological record pointer for a KSAM disc file. The file may contain either fixed-length or variable-length records. When the next FREADC request is issued for this file, the record to which FPOINT positioned the pointer is read. Note that this intrinsic positions the logical record pointer as well as the chronological pointer.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file on which the pointer is to be set.

recnum *double by value (required)*

A positive double integer representing the record number of a fixedlength file or the word pointer to a variable-length file. Word numbering always starts with word 0, whereas record numbering starts with 0 or 1 depending on how the file was created. In either case, the number is in terms of the chronological (consecutive) order in which the data file records were written. It has no relation to the logical record pointer that is based on key values.

CONDITION CODES

CCE Request granted.

CCG Request denied. The chronological record pointer position is unchanged. Positioning was requested at a point beyond the physical end-of-file.

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

Invalid *filenum* parameter.
recnum parameter specified a record marked for deletion.
A key value for specified record not found in key file.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FPOINT

The FPOINT intrinsic is generally used prior to an FREADC intrinsic in order to read a record without reference to the key file. FPOINT sets the chronological record pointer to the position in the file specified by *recnum*. A subsequent FREADC reads the record (or portion of a record) to which the pointer is positioned. It then sets the pointer to the next record that was written to the file in chronological order,

For example, in order to read the 39th record written to the file identified by FILENUM:

```
FPOINT(FILNUM,39D); <<set pointer>>
FREADC(FILNUM,BUFFER,COUNT); <<read record>>
```

Following execution of FREADC, the contents of the 39th record are transferred to the array BUFFER and the chronological pointer remains positioned at record 39. A flag is set so that the next call to FREADC moves the pointer forward to the beginning of record 40, the next record in chronological order.

Note that the combination of FPOINT followed by an FREADC intrinsic is identical in effect to the FREADDIR intrinsic that positions to a chronological record number and then reads that record. The FGETINFO intrinsic can be used to recover the chronological record number of the record most recently accessed. (Refer to FGETINFO and FREADDIR for more information on accessing records by chronological record number.)

Since the FPOINT intrinsic positions the logical pointer as well as the chronological pointer, it can be used prior to an FUPDATE or FREAD intrinsic to identify the record to be updated or read. FPOINT sets the logical record pointer to a key in the key file that points to the record it located by record number. The key is by default the primary key for that record, though an alternate key is used if such a key was selected by a prior call to FFINDBYKEY or FREADBYKEY.

SHARED ACCESS

When you use FPOINT to position the chronological pointer in a shared access environment, you must lock the file with a call to FLOCK before calling FPOINT. You should leave the file locked until you have completed any calls that read or update the file in chronological sequence, and then call FUNLOCK to unlock the file for the other users. This insures that the pointer is not moved by other users between the pointer-independent procedure FPOINT and any subsequent pointer-dependent procedure. (Refer to Table 4-2 for a list of the pointer-independent and pointer-dependent procedures.)

FREAD

INTRINSIC NUMBER 2

Reads a logical record in key sequence from a KSAM file to the user's stack.

```
    I                IV    LA        IV
lgth:=FREAD(filenum,target,tcount);
```

FREAD reads a logical record in sequential order by key value. The primary key determines key sequence unless a prior call to FFINDN (or FFINDBYKEY or FREADBYKEY) has specified an alternate key. If the file is opened without KSAM access (FOPEN *aoptions* bit 3=1), then FREAD reads the data file as if it were not a KSAM file.

The record read by FREAD depends on the current position of the logical record pointer.

FUNCTIONAL RETURN

The FREAD intrinsic returns a positive integer value to *lgth* showing the length of the information transferred. If the *tcount* parameter in the FREAD call is positive, the positive value returned represents a *word* count; if the *tcount* parameter is negative, the positive value returned represents a *byte* count.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be read.

target *logical array (required)*

An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.

tcount *integer by value (required)*

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in *words*; if it is negative, it signifies the length in *bytes*; if it is zero, no transfer occurs.

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

CONDITION CODES

CCE The information was read.

CCG The logical end-of-data was encountered during reading.

CCL The information was not read because an error occurred.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FREAD

The FREAD intrinsic reads the record at which the logical record pointer is currently positioned. When a file is opened, this pointer is positioned to the beginning of the first record in primary key sequence. That is, it is positioned to the record containing the lowest value in those bytes containing the primary key.

Following each FREAD , the record pointer remains positioned at the record just read. Any subsequent FREAD call positions the pointer to the next sequential record in ascending key sequence. Also, if an FREAD call is followed by an FUPDATE and another FREAD, the pointer is advanced before the second FREAD.

A key other than the primary key can be selected as the basis of the sequential read by executing FFINDN, FFINDBYKEY, or FREADBYKEY before executing the FREAD intrinsic.

When the logical end-of-data is encountered during reading, the CCG condition code is returned to your process. The end-of-data occurs when the last logical record of the file is passed . Note that the last logical record of a KSAM file is the record containing the maximum key value in the key on which the key sequence is based.

SHARED ACCESS

In order to be sure that you are reading the record you want, you should call either FLOCK or FCONTROL with control code 7 before calling FREAD. FLOCK prevents other users from changing or deleting the record until the file is unlocked with FUNLOCK. FCONTROL with control code 7 clears the data and key block buffers so that the record must be read directly from the file, and also transfers the latest control information from the file to the extra data segment. Because the logical pointer is part of this control information, you can be sure that is is set correctly by calling FCONTROL with code 7.

FCONTROL uses less overhead than FLOCK, but it cannot prevent other users from modifying the record you want to read while you are calling FCONTROL. FLOCK, on the other hand, fully protects the information to be read from changes by other users but requires more time.

Because FREAD is a pointer-dependent procedure, you must call one of the procedures that position the pointer before calling FREAD. When you are reading the file in sequential key order, it is important to lock the file before calling the procedure that positions the pointer, and to leave it locked while you are reading the file. This insures that the pointer is not moved by another user between the call that positions the pointer and FREAD or between sequential FREAD calls. (Refer to Table 4-2 for a list of the pointer-independent and pointer-dependent procedures.)

For example, the following sequence of calls guarantees that you will read the file in sequential order starting with a specified key:

```
FLOCK
FFINDBYKEY <<sets logical pointer>>
   FREAD loop <<read records in key sequence>>
FUNLOCK
```

Note that FREAD advances the record pointer only if it is followed by another FREAD (or an FUPDATE followed by another FREAD). A single call to FREAD leaves the pointer at the record just read; a subsequent call to FREAD causes the pointer to be positioned to the next record in key sequence. This permits sequential reading of the file without calling a pointer-independent procedure before each FREAD. Also, in order to allow sequential updates, the pointer is advanced for each FREAD in an FUPDATE/FREAD sequence with no other intervening calls (see FUPDATE discussion).

In the example in Figure 4-7, FREAD is used first to read the KSAM file in sequence by primary key. When the end of data is reached, the program uses FFINDBYKEY to specify an alternate key and FREAD then reads the file in sequence by that altemate key. When the end of data is reached again, the file is closed. (Note that this program is opened for exclusive access so that locking is not necessary).

```
$CONTROL MAIN=JEXAMPL2                                          READ SEQUENTIALLY BY PRI-
<<*********************************************************>>  MARY KEY
<<*                                                       *>>
<<*                    EXAMPLE 2                          *>>
<<*          READ A KSAM FILE SEQUENTIALLY                *>>
<<*                                                       *>>
<<*********************************************************>>
  INTEGER      FILNUM;
  INTEGER ERRORCODE,LENGTH;
  BYTE ARRAY   FILNAME(0:9):="JEXAMFIL ";
ARRAY          MESSAGE(0:35);
ARRAY          INPUT(0:39);
ARRAY          OUTPUT(*)=INPUT;
  BYTE ARRAY    KEYVALUE(0:7:="000-0000";
  INTEGER       KEYLENGTH"=8;
  INTEGER       KEYLOCATION:21;
  INTEGER       RELOP:2;
  INTRINSIC   FOPEN,FCLOSE,  FREAD,FFINDBYKEY  ,READ,PRINT,
          FCHECK,FERRMSG,PRINT'FILE'INFO,TERMINATE;
<<*************************>>
<<* OPEN THE KSAM FILE    *>>
<<*************************>>
FILENUM:=FOPEN(FILNAME,3,200); <<OPEN KSAM FILE
                  FOR EXCLUSIVE READ-ONLY ACCESS>>
IF FILNUM=0
THEN BEGIN           <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE);          <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH); <<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0);    <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
MOVE MESSAGE:="LIST IN LAST NAME SEQUENCE";
PRINT(MESSAGE,-26,0);
<<*********************************>>
<<* READ KSAM IN NAME SEQUENCE     *>>
<<*********************************>>
L1:
  FREAD(FILNUM,INPUT,-72);
  IF   >                                                        GO TO ALTERNATE KEY ORDER
  THEN GO TO L2;
```

Figure 4-7. FREAD Example

```
IF <
THEN BEGIN
       MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
       PRINT(MESSAGES,-34,0);
       TERMINATE;
     END;
<<*********************************************>>
<<* WRITE THE DATA JUST READ FROM KSAM FILE     *>>
<<*********************************************>>
PRINT(OUTPUT,-72,0);
<<*********************************>>
<<* GO BACK TO GET ANOTHER RECORD   *>>
<<*********************************>>
GO TO L1;
<<*********************************************************>>
<<* READ DATA FROM KSAM FILE IN TELEPHONE # SEQUENCE      *>>
<<*********************************************************>>
  L2:
  FFINDNYKEY(FILNUM,KEYVALUE,KEYLOCATION,KEYLENGTH,RELOP);
  MOVE MESSAGE:="LIST IN TELEPHONE NO. SEQUENCE";
  PRINT(MESSAGE,-30,0);
  L3:
  FREAD(FILNUM,INPUT,-72);
READ SEQUENTIALLY BY ALTERNATE KEY
IF >
THEN BEGIN <<END OF FILE>>
       FCLOSE(FILNUM,0,0); <<CLOSE THE KSAM FILE>>
       IF <> THEN
         BEGIN <<CLOSE UNSUCCESSFUL>>
            MOVE MESSAGE:"CANNOT CLOSE THE KSAM FILE":
            PRINT(MESSAGE,-29,0);
            FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
            FERRMSG(ERRORCODE,MESSAGE,LENSTH;<<GET MESSAGE STRING>>
            PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
          END;
        TERMINATE;
      END;
 IF <
 THEN BEGIN
        MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
        PRINT(MESSAGE,-34,0);
        TERMINATE;
      END;
```

Figure 4-7. FREAD Example (continued)

```
<<*********************************************>>
<<* WRITE THE DATA JUST READ FROM KSAM FILE     *>>
<<*********************************************>>
PRINT(OUTPUT,-72,0);
IF <>
THEN BEGIN <<ERROR OCCURRED WHILE PRINTING OUTPUT>>
       MOVE MESSAGE,="ERROR OCCURRED WHILE PRINTING OUTPUT";
       PRINT(MESSAGE,-36,0);
       FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH),<<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*************************************>>
<<* GO BACK TO GET ANOTHER RECORD    *>>
<<*************************************>>
GO TO L3;
END;

Output from Program Execution:

  LIST IN LAST NAME SEQUENCE
  CARDIN RICK 578-7018 11100 WOLFE ROAD CUPERTINO CA. 94053
  ECKSTEIN LEO 287-5137 5303 STEVENS 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 BERKELEY CA. 90871
  SEELY HENRY 293-4220 1144 LEBERTY ST. EL CERRITO CA. 94053
  TURNEWR IVAN 984-8498 22905 EMERSON ST. OAKLAND CA. 98234
  WESTER ELDER 287-4598 1256 KINGFISHER SUNNYVALE CA. 43098
  WHITE GORDON 398-0301 4350 ASHBY AVE. BERKELEY CA. 91234
  LIST IN TELEPHONE NO. SEQUENCE
  HOSODA JOE 227-8214 1180 SAINT PETER CT. LOS ALTOS CA. 94022
  ROBERT GERRY 259-5535 12345 TELEGRAPH BERKELEY CA. 90871
  WESTER ELDER 267-4598 1256 KINGFISHER SUNNYVALE CA. 43098
  ECKSTEIN LEO 287-5137 5303 STEVENS SANTA CLARA CA. 95050
  SEELY HENRY 293-4220 1144 LEBERTY 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 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
```

Figure 4-7. FREAD Example (continued)

FREADBYKEY

INTRINSIC NUMBER 304

Reads a logical record randomly from a KSAM file to the user's data stack.

```
      I                       IV     LA       IV       BA         IV
lgth:=FREADBYKEY(filenum,target,tcount,keyvalue,keylocation);
```

FREADBYKEY reads a logical record selected by key value. The record to be read must have the same value as *keyvalue* in the bytes that start at *keylocation*. Following execution, the logical record pointer is still positioned to the record in the file located through the value of the key at *keylocation*.

FUNCTIONAL RETURN

The FREADBYKEY intrinsic returns a positive integer value to *lgth* showing the length of the information transferred. If the *tcount* parameter in the FREADBYKEY call is positive, the positive value returned represents a word count; if the *tcount* parameter is negative, the positive value returned represents a byte count.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be read randomly.

target *logical array (required)*

An array to which the record is to be transferred. It should be large enough to hold all the information read.

tcount *integer by value (required)*

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in words; if negative, it signifies the length in bytes; if zero, no transfer takes place.

If *tcount* is less than the size of the record, only the first *tcount* words are read from the record. If *tcount* is larger than the physical record size, transfer is limited to the length of the physical record.

keyvalue *byte array (required)*

A byte array containing the value that will determine which record is read. The first record found with this identical value in the key identified by *keylocation* is the record read.

keylocation *integer by value (required)*

The relative byte location in the record of the key whose value determines which record is read. The first byte is numbered 1; if a value of zero is specified, the primary key is used.

CONDITION CODES

CCE	The information specified was read.
CCG	The logical end-of-data or beginning-of-data was encountered during the read.
CCL	The information was not read because an error occured, such as an input/output error, or the key could not be located.

USING FREADBYKEY

The intrinsic FREADBYKEY allows you to locate and read a single record according to a specified key value, Like FFINDBYKEY, it defines the key that is to be used for determining record sequence and, following execution, remains positioned at the same record. *Unlike FFINDBYKEY, FREADBYKEY cannot specify a key length different from the full length of the key at creation, nor can it search for approximate key values.*

In the example in Figure 4-8, the *keylocation* and *keyvalue* values are read from the standard input device. As each is read, it is printed to test the read. The first set of values read into the word array INFOW is:

```
01 ROBERT GERRY
\/\------------/
|             |
%%keylocation%%  %%keyvalue%%
```

The first two ASCII characters contain the *keylocation*; the characters starting in byte 2 contain the *keyvalue* to be found at the specified *keylocation*. Since *keylocation* is an integer parameter, the first two bytes of the byte array INFO (equivalenced to the word array INFOW) must be converted to a binary value. This is done with the statement:

```
KEYLOCATION:=BINARY(INFO,2);
```

The value to be used for *keyvalue* is contained in the byte array INFO starting in the third byte (byte 2 numbered from byte 0). In the declarations at the beginning of the program, the byte array KEYVALUE is equivalenced to the portion of the byte array INFO that starts in byte 2.

The intrinsic FREADBYKEY can be called with the following statement:

```
FREADBYKEY(FILNUM,INPUT,-72,KEYVALUE,KEYLOCATION);
```

This locates and reads the first record with the value ROBERT GERRY in the key located starting in byte 1 of the record. The program prints this record and then returns to get the next pair of values input for *keyvalue* and *keylocation*. When there are no more values in the input file, the KSAM file is closed and the program terminates.

SHARED ACCESS

If you use FREADBYKEY to position the pointer for subsequent calls that read or update the specified record, you should lock the file with a call to FLOCK before calling FREADBYKEY. Then, after calling the read or update procedure, you should unlock the file so other users can access it. Locking the file before calling FREADBYKEY insures that other users do not change the position of the pointer between the call to FREADBYKEY and any subsequent procedure that depends on the pointer position. (Refer to Table 4-2 for a list of the pointerpendent procedures and also those that set the pointer.)

To illustrate, the following sequence of calls makes sure that the correct record is updated:

```
FLOCK        to lock the file
FREADBYKEY   to position the pointer
FUPDATE      to modify the record to which the pointer points
FUNLOCK      to unlock the file for other users
```

DUPLICATE KEYS

FREADBYKEY always positions to the first key in a chain of duplicate keys. If you want to read or update the remaining keys in a duplicate key chain, you should use FREAD. For example, to update all the records with a particular key, use the following code sequence:

```
FREADBYKEY   to locate 1st key in chain of duplicates
FUPDATE      update that record
FREAD        read next sequential record
             test if this is correct key value
FUPDATE      update record
             return to read next record
```

```
$CONTROL MAIN=JEXAMPL3
<<*********************************************************>>
<<*                                                       *>>
<<*                   EXAMPLE 3                           *>>
<<*             READ A KSAM FILE RANDOMLY                 *>>
<<*                                                       *>>
<<*********************************************************>>
INTEGER              FILNUM;
INTEGER              ERRORCODE,LENGTH;
BYTE ARRAY           FILNAME(0:9):="JEXAMFIL ";
ARRAY                MESSAGE(0:35);
ARRAY                INPUT(0:39);
ARRAY                OUTPUT(*)=INPUT;
BYTE ARRAY           INFO(0:35);
ARRAY                INFOW(*)=INFO
BYTE ARRAY           KEYVALUE(*)INFO(2);
INTEGER              KEYLOCATION;
INTRINSIC FOPEN,FCLOSE,FREAD,FREADBYKEY,READ,PRINT,
          FCHECK,FERRMSG,BINARY,TERMINATE;
<<*********************************************************>>
<<*           OPEN THE KSAM FILE                          *>>
<<*********************************************************>>
FILNUM:=FOPEN(FILNAME,3); <<OPEN THE KSAM FILE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:"CANNOT OPEN KSAM FILE"
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<GET MESSAGE STRING>>
       PRINTIMESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*********************************************************>>
<<*      READ IN KEYVALUE AND KEYLOCATION INFOMATION      *>>
<<*********************************************************>>
L1:
READ(INFOW,-36);
IF >
THEN BEGIN
       FCLOSE(FILNUM,0,0); <<CLOSE THE KSAM FILE>>
       IF <> THEN
          BEGIN
            MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE"
            PRINT (MESSAGE,-26,0);
```

Figure 4-8. FREADBYKEY Example

```
            FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
            FERRMSG(ERRORCODES;MESSAGE,LENGTH);<<GET MESSAGE STRING>>
            PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
          END;
       TERMINATE;
     END;
IF <
THEN BEGIN
         MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
         PRINT(MESSAGE,-34,0);
         TERMINATE;
PRINT(INFOW,-36,0); <<TEST READ>>
KEYLOCATION;=BINARY(INFO,2); <<CONVERT FROM ASCII TO BINARY>>
<<*****************************************************>>
<<*   READ KSAM ACCORDING TO KEYVALUE AND KEYLOCATION *>>
<<*****************************************************>>
FREADBYKEY(FILNUM,INPUT,-72,KEYVALUE,KEYLOCATION);
IF <>
THEN BEGIN <<ERROR OCCURRED IN FREADDBYKEY>>
        MOVE MESSAGE:="ERROR OCCURRED IN FREADBYKEY";
        PRINT(MESSAGE,-28,0);
        FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
        FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<GET MESSAGE STRING>>
        PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
        GO TO L1;
     END;
<<*************************************************>>
<<*      WRITE THE DATA JUST READ FROM KSAM FILE *>>
<<*************************************************>>
PRINT(OUTPUT,-72,0);
<<*********************************>>
<<* GO BACK TO GET ANOTHER RECORD *>>
<<*********************************>>
GO TO L1;
END;

Output from Program Execution:

01ROBERT GERRY
ROBERT GERRY 259-5535 12345 TELEGRAPH AVE. BERKELEY CA. 90871
21287-5137
ECKSTEIN LEO 287-5137 5303 STEVENS CREEK SANTA CLARA CA. 95050

 END OF PROGRAM
```

Figure 4-8. FREADBYKEY Example (continued)

FREADC

INTRINSIC NUMBER 305

Reads a logical record in chronological sequence from KSAM file to user's stack.

```
   I              IV      LA     IV
lgth:=FREADC(filenum,target,tcount);
```

FREADC reads a logical record in chronological sequence. Chronological sequence means the sequence in which the records were originally written to the data file.

When FREADC is executed, the key file is not accessed. This read is similar to the standard FREAD for non-KSAM files except that FREADC skips any data records that are marked for deletion. Following execution, the chronological pointer remains positioned at the same record.

FUNCTIONAL RETURN

The FREADC intrinsic returns a positive integer value to *lgth* showing the length of the information transferred. If the *tcount* parameter in the FREADC call is positive, the positive value returned represents a *word* count; if the *tcount* parameter is negative, the positive value returned is a *byte* count.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be read in chronological sequence.

target *logical away (required)*

An array to which the record is to be transfered. This array should be large enough to hold all the information to be transferred.

tcount *integer by value (required)*

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in words; if negative, it signifies the length in bytes; if zero, no transfer occurs.

If *tcount* is less than the size of the record, only the first *tcount* words are transfered from the record. If *tcount* is larger than the physical record size, transfer is limited to the length of the physical record.

CONDITION CODES

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

USING FREADC

This intrinsic allows you to read the records in the data file in the order in which they are physically stored in the file. The end-of-data is encountered following the last record in the file. If any records have been marked for deletion (refer to the FREMOVE intrinsic), these records are not read; otherwise, this intrinsic reads the data from the data file exactly as it was stored.

Following execution of FREADC, the chronological pointer remains positioned at the record just read, unless it is followed by another call to FREADC. In a series of calls to FREADC, the pointer is advanced automatically so you can read the file in chronological sequence without resetting the pointer for each record.

Because FUPDATE only checks the logical pointer, you cannot update a record located by FREADC or FREADDIR. To update a record located by its chronological record number, you must precede the call to FUPDATE with a call to FPOINT. Unlike FREADC or FREADDIR, FPOINT sets the logical pointer as well as the chronological pointer.

In Figure 4-9, the FREADC intrinsic is used to read the data from the KSAM data file in chronological order. Compare this order to the sequential order by primary key in which the same file is read by FREAD. (Refer to Figure 4-10 for an example showing the chronological record number printed in association with each record listed in sequential key order.)

SHARED ACCESS

Because FREADC is a chronological pointer-dependent procedure, you must call one of the procedures that position the pointer before calling FREADC. (Refer to Table 4-2 for a list of the pointer-dependent and pointer-independent procedures.) When access is shared, it is essential that you lock the file before calling the procedure that positions the pointer, and then leave the file locked while it is being read by FREADC. This insures that no other user changes the information in the record.

For example, the following sequence of calls guarantees that you will read the file in chronological sequence starting with a specified record number:

```
FLOCK <----------- lock file
FPOINT <------------- position the chronological pointer
  FREADC loop <----------------- read records in chronological sequence
FUNLOCK <--------------- unlock file
```

```
$CONTROL MAIN=JEXAMPL4
<<*********************************************************>>
<<*                                                       *>>
<<*                  EXAMPLE 4                            *>>
<<*      READ A KSAM FILE CHRONOLOGICALLY                 *>>
<<*                                                       *>>
<<*********************************************************>>
INTEGER FILNUM;
INTEGER LENGTH;
INTEGER ERRORCODE;
BYTE ARRAY FILENAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE (0:35);
ARRAY INPUT(0:39);
ARAY OUTPUT(*)=INPUT;
INTRINSIC FOPEN,FCLOSE,FREADC,FCHECK,FERRMSG,PRINT,TERMINATE;
<<*************************>>
<<*  OPEN THE KSAM FILE  *>>
<<*************************>>
FILNUMI=FOPEN(FILNAME,3); <<OPEN THE KSAM FILE>>
IF FILNUM=0
THEN BEGIN
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCIDE); <<GET THE ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH):<<GET MESSAGE STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
       TERMINATE;
     END;
L1;
<<*****************************************************>>
<<*    READ KSAM ACCORDINGTO CHRONOLOGICAL ORDER    *>>
<<*****************************************************>>
FREADC(FILNUM,INPUT,-72);
IF >
THEN BEGIN <<END OF FILE>>
       FCLOSE(FILNUM,0,0); <<CLOSE THE KSAM FILE>>
       IF <>
       THEN BEGIN
             MOVE MESSAGE:="CANNOT CLOSE KSAM FILE";
             PRINT(MESSAGE,-22,0);
             FCHECK(FILNUM,ERRORCODE); <<GET THE ERROR NUMBER>>
             FFERRMSG(ERRORCODE,MESSAGE,LENGTH);<<GET MESSAGE STRING>>>
             PRINT(MESSAGE,-LENGTH,0); <<PRINT ERROR MESSAGE>>
           END;
       TERMINATE;
    END;
```

Figure 4-9. FREADC Example

```
IF <
THEN BEGIN
       MOVE MESSAGE:="ERROR OCCURRED WHILE READING KSAM FILE";
       PRINT(MESSAGE,-37,0);
       TERMINATE;
      END;
<<***********************************************>>
<<* WRITE THE DATA JUST READ FROM KSAM FILE *>>
<<***********************************************>>
PRINT(OUTPUT,-72,0);
<<*************************************>>
<<* GO BACK TO GET ANOTHER RECORD *>>
<<*************************************>>
GO TO L1;
END;

Output from Program 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 578-7015 11100 WOLFE ROAD CUPERTINO CA. 94053
PASBY LINDA 295-1187 TOWN & CNTRY VILLAGE SAN JOSE CA. 94102
SEELY HENRY 293-4220 1144 LEBERTY ST. EL CERRITO CA. 94053
ROBERT GERRY 259-5535 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
```

Figure 4-9. FREADC Example (continued)

FREADDIR

INTRINSIC NUMBER 7

Reads a logical record located by its chronological record number from a KSAM file to the user's stack.

```
                    IV    LA    IV      DV
FREADDIR(filenum,target,tcount,recnum);
```

The FREADDIR intrinsic reads a specific logical record, or a portion of such a record, from a KSAM file to the user's data stack. The particular record read is specified by its chronological record number. This number is determined by the order in which the record was written to the file; it is not the logical record number determined by ascending key sequence. When the file has fixed-length records, *recnum* is the actual record number counting from the first record in the file. When the file has variable-length records, *recnum* is a word pointer to the first word in the record counting from the first word in the file, word zero.

After FREADDIR has been executed, the chronological record pointer remains positioned at the record just read. FREADDIR does not change the position of the logical record pointer.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file to be read.

target *logical array (required)*

An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.

tcount *integer by value (required)*

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies *words*; if negative, it signifies *bytes*; and if it is zero, no transfer occurs.

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

recnum *double by value (required)*

A double-word integer indicating the relative chronological record number (or word number for variable-length records) to which the chronological pointer is positioned. Chronological record numbering for fixed-length records starts with zero or one, as specified in *ksamparam* or by FIRSTREC in BUILD.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FREADDIR

You can use the FREADDIR intrinsic to position to a particular record in chronological sequence and then read that record. Following execution, the record pointer remains positioned at the same record. This intrinsic is similar in effect to the pair of intrinsics FPOINT and FREADC, with one exception: FREADDIR does not skip records marked for deletion. You might use FREADDIR to read one record and then reposition the pointer; you might use FPOINT and FREADC to position to a particular record and then continue reading in chronological order from that position.

You can use the FGETINFO intrinsic to determine the relative chronological number of the record most recently accessed. This number is returned in the FGETINFO parameter *recpt*. The example in Figure 4-10 determines the chronological record of each record as it is read in sequence by primary key value. The chronological record number is printed, and then FREADDIR uses this number to read the record to which it points. The output shows the chronological record number followed by the record to which it points. To see these records listed in chronological order, refer to the output in example 4-9 illustrating FREADC.

Note that execution of those intrinsics that position the KSAM data file by means of the chronological record pointer (FPOINT, FREADC, FREADDIR), do not access the key file. This type of access only affects the data file. It is, therefore, much faster than those intrinsics that use key sequence to position the data file and must access the key file.

```
$CONTROL MAIN=JEXAMPL6
<<*****************************************************>>
<<*                                                   *>>
<<*                   EXAMPLE 6                       *>>
<<*   READ A KSAM FILE BY CHRONOLOGICAL RECORD NUMBER  *>>
<<*                                                   *>>
<<*****************************************************>>
INTEGER FILNUM;
INTEGER ERRORCODE;
INTEGER LENGTH;
BYTE ARRAY FILNAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
DOUBLE RECPTR;
INTRINSIC FOPEN,FCLOSE,FREAD,FGETINFO,FREADDIR,
          PRINT,TERMINATE,DASCII,FCHECK,FERRMSG;
<<*************************>>
<<* OPEN THE KSAM FILE *>>
<<*************************>>
FILNUM,=FOPEN(FILNAME,3); <<OPEN THE KSAM FILE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*****************************************************>>
<<*              READ KSAM SEQUENTIALLY               *>>
<<****************#************************************>>
L1:
FREAD(FILNUM,INPUT,-72);
IF >
THEN BEGIN
        FCLOSE(FILNUM,0,0); <<CLOSE THE KSAM FILE>>
        IF <> THEN
            BEGIN
            MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE";
            PRINT(MESSAGE,-22,0);
```

Figure 4-10. FREADDIR Example

```
            FCHECK(FILNUM,ERRORCOOE); <<GET ERROR NUMBER>>
            FFRRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
            PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
          END;
        TERMINATE;
      END;
IF <
THEN BEGIN
       MOVE MESSAGE:="ERROR OCCURRED WHILE READING KSAM FILE";
       PRINT(MESSAGE,-37,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
              TERMINATE;
            END;
      <<*****************************************************>>
      <<* TO FIND OUT RECORD NUMBER OF THE RECORD JUST READ *>>
      <<*****************************************************>>
      FGETINFO(FINUM,,,,,,,RECPTR);
           MOVE MESSAGE:="RECORD# = ";
           DASCII(RECPTR,10,MESSAGE(5));
           PRINT(MESSAGE,-14,0);
      <<*********************************************>>
      <<* READ THE KSAM FILE BY USING RECORD NUMBER *>>
      <<*********************************************>>
      FREADDIR(FILNUM,INPUT,-72,RECPTR);
      IF <>
      THEN BEGIN
             MOVE MESSAGE:="ERROR OCCURRED DURING FREADDIR";
             PRINT(MESSAGE,-30,0);
             FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
             FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>>
             PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
             TERMINATE;
           END;
      <<*********************************************>>
      <<* WRITE THE DATA JUST READ BY FREADDIR *>>
      <<*********************************************>>
      PRINT(OUTPUT,-72,0);
      <<*********************************>>
      <<* GO BACK TO GET ANOTHER RECORD *>>
      <<*********************************>>
      GO TO L1;
      END;
```

Figure 4-10. FREADDIR Example (continued)

```
Output from Program Execution:

RECORD# = 4
CARDIN RICK 578-7018 11100 WOLFE ROAD CUPERTINO CA. 94053
RECORD# = 3
ECKSTEIN LEO 287-5137 5303 STEVENS CREEK SANTA CLARA CA. 95050
RECORD# = 2
HOSODA JOE 227-8214 1180 SAINT PETER CT. LOS ALTOS CA. 94022
RECORD# = 1
NOLAN JACK 923-4975 967 REED AVE. SUNNYVALE CA. 94087
RECORD# = 5
PASBY LINDA 295-1187 TOWN & CNTRY VILLAGE SAN JOSE CA. 94102
RECORD# = 7
ROBERT GERRY 259-5535 12345 TELEGRAPH AVE. BERKELEY CA. 90871
RECORD# = 6
SEELY HENRY 293-4220 1144 LEBERTY ST. EL CERRITO CA. 94053
RECORD# = 8
TURNEWR IVAN 984-8498 22905 EMERSON ST. OAKLAND CA. 98234
RECORD# = 10
WESTER ELDER 287-4598 1256 KINGFISHER ST. SUNNYVALE CA. 43098
RECORD# = 9
WHITE GORDON 398-0301 4350 ASHBY AVE. BERKELEY CA. 91234

 END OF PROGRAM
```

Figure 4-10. FREADDIR Example (continued)

FREADLABEL

INTRINSIC NUMBER 19

Reads a user file label.

```
                IV        LA      IV      IV     O-V
FREADLABEL(filenum,target,tcount,labelid);
```

The FREADLABEL intrinsic reads a user-defined label from a disc file. Before reading occurs, the user's read-access capability is verified. Note that MPE automatically skips over any unread user labels when the first FREAD intrinsic call is issued for a file; therefore the FREADLABEL intrinsic should be called immediately after the FOPEN intrinsic has opened the file.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file whose label is to be read.

target *logical array (required)*

An array in the stack to which the label is to be transferred. This array should be large enough to hold the number of words specified by *tcount*.

tcount *integer by value (optional)*

An integer specifying the number of words to be transferred from the label. *Tcount* must not be greater than 128 words.

Default: 128 words.

labelid *integer by ualue (optional)* An integer specifying the label number where the first user label is numbered 0.

Default: A default value of 0 is assigned.

CONDITION CODES

CCE	The label was read
CCG	The intrinsic referenced a label beyond the last label written on the file.
CCL	The label was not read because an error occurred.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FREADLABEL

If the KSAM file contains one or more user labels (written with FWRITELABEL), you can read these labels with the FREADLABEL intrinsic. During the normal file reads with FREAD, FREADC, FREADBYKEY, or FREADDIR, any user labels are ignored. The number of user labels that can be written to the file is specified by the *userlabels* parameter of FOPEN, or in the BUILD command of KSAMUTIL.

Since MPE checks to insure that you have opened the file with read access before executing FREADLABEL, you must open the file with an FOPEN *aoptions* setting that permits reading. It must be one of the following:

bits 12:4 = 0000 (octal 0) read only access

= 0100 (octal 4) input/output access

= 0101 (octal 5) update access

In addition, the FOPEN *userlabels* parameter must be set to a value of 1 or greater depending on the number of labels that may be written to the file.

Suppose you have opened the file KDATA with the following call:

```
KFILNUM:=FOPEN(KDATA,3,4,,,,2);
                        / ^    \
                       /  |     \
                      /   |   number of labels
              old user/   |
              domain      |
                          |
                          |
                          |
                          input/output access
```

You might read the second label with the following call:

```
FREADLABEL(KFILNUM,LABELBUF,,1)
```

This reads the second label into the array LABELBUF. Note that label numbering begins with 0; if the labelid parameter were zero or omitted, then the first label would be read. By default, the number of words read from the label is 128.

FREADSEEK

INTRINSIC NUMBER 12

Moves a record from a disc file to a buffer in anticipation of a FREADDIR intrinsic call.

Note This intrinsic may not be used for KSAM files. If called for a file created as a KSAM file, the intrinsic returns a CCL condition code.

FRELATE

INTRINSIC NUMBER 18

Determines whether a file pair is interactive, duplicative, or both interactive and duplicative.

Note This intrinsic may not be used for KSAM files. If called for a file created as a KSAM file, the functional return is set to zero (FALSE) and the condition code CCE is returned.

FREMOVE

INTRINSIC NUMBER 306

Marks the current record in KSAM file for deletion.

```
                    IV
FREMOVE(filenum);
```

The intrinsic FREMOVE effectively removes the current record from the KSAM file. When executed, the first two characters of the current record in the data file is set to all 1's, and all key entries pointing to this record are deleted from the key file. Although the space required by the record remains in the data file, it is no longer possible to access the record through KSAM intrinsics.

In order to position the file to the record to be deleted, FREMOVE must be preceded by one of the intrinsics that positions the logical record pointer: FFINDN, FFINDBYKEY, FREADBYKEY, FREAD, FPOINT, or a previous FREMOVE. Following execution of FREMOVE, the logical record pointer is positioned at the next record in ascending key sequence.

FREMOVE checks only the logical record pointer, not the chronological pointer, to locate the record to be deleted. Therefore, if you want to delete a record located by its chronological position in the file, precede the call to FREMOVE with a call to FPOINT. FPOINT locates the record by its record number and sets the logical, as well as the chronological pointer, to that record. If you try to locate a record for FREMOVE by calling FREADDIR or FREADC, which only set the chronological pointer, you will delete the wrong record.

When FREMOVE is executed, a check is made to make sure the record to be deleted actually contains the key value to which the pointer is positioned. If the record does not contain that value, then a condition code (CCL, error=191) is issued and the record is not deleted.

If the file was opened for shared access (*aoptions* bits 8,9 = 11) then you must call FLOCK before calling FREMOVE. Note that the file must also have been opened with dynamic locking allowed (*aoptions* bit 10 = 1).

Note If you want to recover the data in deleted records through nonKSAM access (using FCOPY with the NOKSAM option), do not place any data in the first two bytes since these bytes are overwritten by FREMOVE.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file from which the record is to be deleted.

CONDITION CODES

CCE	The current record is deleted.
CCG	The logical end-of-data was encountered.
CCL	An error was encountered or record does not contain requested key value; the record is not deleted.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FREMOVE

When FREMOVE is executed, it sets the first word (bytes 1 and 2) of the current record to all 1's. It does not physically delete the record from the file. When the file is read by any of the KSAM read intrinsics, the deleted records are skipped as if they were not there. Since all references to them are deleted from the key file, the speed of execution is not usually affected by the records physically remaining in the file. However, they do take up space and if a great many records are deleted, then you might want to build a new KSAM file and copy the old file to the new file with FCOPY. Since FCOPY does not copy records marked for deletion (except with the NOKSAM option), the new file will be shorter and have no space used by deleted records. (Refer to section II for a description of copying KSAM files with FCOPY.)

The example in Figure 4-11 deletes all records with a telephone number in the alternate key field that is equal to or greater than "500-0000". The FFINDBYKEY intrinsic positions the file to the record containing the lowest alternate key value that is greater than or equal to "500-0000". This record is then read and printed prior to being deleted by FREMOVE. Following FREMOVE, the program loops back to read the next sequential record, print it, and then delete it. When an end of data is reached, the program terminates. In all, the program deletes three records. You can check the deleted records against the list of records printed in telephone number sequence by the program illustrating FREAD in Figure 4-7.

In practice, an FREAD prior to an FREMOVE is useful because it allows you to test the record contents prior to deleting the record. For example, you might want to delete only those records with the zip code 90871 in bytes 75 through 79 of the record, assuming the same file as in Figure 4-11:

```
BYTE ARRAY INPUTB(*)=INPUT; <---- Equate the byte array INPUTB to INPUT
:
L1:
FREAD(FILNUM,INPUT,-72);
IF>
    .
    . <----- test for end of data
    .
IF<
    .
    .<------ test for read error
    .
IF INPUTB(75)="90871"
THEN BEGIN
        FREMOVE(FILNUM);
       IF<
             .
             . <----- test for delete error
             .
         END;
GO TO L1; <--------------- return for next record
```

SHARED ACCESS

In a shared environment, you must always lock the file with a call to FLOCK before calling FREMOVE. Furthermore, since the logical record pointer must be positioned before the call to FREMOVE, you should lock the file before calling the procedure that positions the pointer. This prevents other users from affecting the pointer position by adding or deleting records between the time you position the pointer and call FREMOVE. The following sequence of calls illustrates the correct method for deleting a record in a shared environment:

```
FLOCK <------------ lock the file
FREADBYKEY <------------ position pointer and read record
FREMOVE <-- mark the record for deletion
FUNLOCK <-- unlock file to allow access to other users
```

Remember to open the file for shared access and allow dynamic locking whenever you plan to delete records from a file in a shared environment.

```
$CONTROL MAIN=JEXAMPL7
<<*******************************************************>>
<<*                    EXAMPLE 7                        *>>
<<*               DELETE SELECTED RECORDS               *>>
<<*******************************************************>>
INTEGER FILNUM;
INTEGER LENGTH;
INTEGER ERRORCODE;
BYTE ARRAY FILNAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
BYTE ARRAY KEYVALUE (0:7):="500-0000";
INTEGER KEYLENGTH:=8;
INTEGER KEYLOCATION;=21;
INTEGER RELOP;=2; << GREATER THAN OR EQUAL TO >>
INTRINSIC FOPEN,FCLOSE,FREAD,FREMOVE,FFINDBYKEY,
          READ,PRINT,TERMINATE,CHECK,FERRMSG;
<<************************>>
<<* OPEN THE KSAM FILE *>>
<<************************>>
FILNUM:=FOPEN(FILNAME,3,5); <<OPEN THE KSAM FILE FOR UPDATE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE:=-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*******************************************************>>
<<* POSITION KSAM FIL IN TELEPHONE # SEQUENCE *>>
<<*******************************************************>>
FFINDBYKEY(FILNUM,KEYVALUE,LEYLOCATION,LEYLENGTH,RELOP);
    MOVE MESSAGE:="DELETE FOLLOWING RECORDS:";
      PRINT(MESSAGE,-25,0);
<<*********************************>>
<<* READ RECORD BEFORE DELETING *>>
<<*********************************>>
L2:
FREAD(FILNUM,INPUT,-72); << READ RECORDS TO BE DELETED>>
```

Figure 4-11. FREMOVE Example

```
IF >
THEN BEGIN <<END OF FILE>>
       FCLOSE(FILNUM,0,0) <<CLOSE THE KSAM FILE>>
       IF <> THEN
         BEGIN <<CLOSE UNSUCCESSFUL>>
           MOVE MESSAGE:="CANNOT CLOSE THE KASM FILE";
           PRINT(MESSAGE,-29,0);
           FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
           FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
           PRINT(MESSGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
         END;
       TERMINATE;
     END;
         IF <
         THEN BEGIN
                MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
                PRINT(MEEAGE,-34,0);
                FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
                FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
                PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
                TERMINATE;
              END;
         <<*********************************************>>
         <<* WRITE THE RECORD JUST READ FROM KSAM FILE *>>
         <<*********************************************>>
         PRINT(OUTPUT,-72,0);
         <<*****************************************>>
         <<* REMOVE RECORD JUST READ FROM FILE *>>
         <<*****************************************>>
         FREMOVE(FILNUM); <<DELETE RECORD>>
         IF <
         THEN BEGIN
                MOVE MESSAGE:=ERORROR OCCURRED DURING DELETE";
                PRINT(MESSAGE,-28,0);
                FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
                FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
                PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
                TERMINATE;
              END;
         <<*********************************>>
         <<* GO BACK TO GET ANOTHER RECORD *>>
         <<*********************************>>
         GO TO L2;
         END,
```

Figure 4-11. FREMOVE Example (continued)

```
Output from Program Execution:

DELETE FOLLOWING RECORDS:
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
```

Figure 4-11. FREMOVE Example (continued)

FRENAME

INTRINSIC NUMBER 17 Renames a disc file.

Note

This intrinsic may not be used for KSAM files. If called for a file created as a KSAM file, the intrinsic returns a CCL condition code.

To rename a KSAM file, use the KSAMUTIL RENAME command.

FSETMODE

INTRINSIC NUMBER 14

Activates or deactivates critical output verification

```
                    IV LV
FSETMODE(filenum,modeflags);
```

The FSETMODE intrinsic activates or deactivates the access mode option that permits critical output verification. This means that all output must be verified as physically complete before control returns from an output intrinsic (FWRITE,FUPDATE, or FREMOVE) to your program.

The access mode established by the FSETMODE intrinsic remains in effect until another FSETMODE call is issued or until the file is closed.

PARAMETERS

filenum *integer by value (required)* A word identifier supplying the file number of the file to which the call applies.

modeflags *logical by value (required)* A 16-bit value that denotes the access mode options in effect. For KSAM files only bit 14 is used; all the remaining bits are set to zeros.

Bit 14=1—; Activate Critical Output Verification When this bit is set, all output to the file is verified as physically complete before an FWRITE, FUPDATE, or FREMOVE intrinsic returns control to the user. As soon as a logical record is written, a CCE condition is returned to the user.

Bit 14=0—; Deactivate Critical Output Verification When the bit is cleared, output is no longer verified.

CONDITION CODES

CCE Request granted.

CCG Not returned by this intrinsic.

CCL Request denied because an error occurred.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FSETMODE

When FSETMODE is executed with the *modeflags* parameter equal to 2 (bit 14=1), then each logical record written by an output intrinsic is physically transferred to the file immediately. Control is not returned to the user program until the transfer has been made. At that time a CCE condition code is returned to the program.

When FSETMODE is executed with the *modeflags* parameter equal to zero (bit 14=0), output is treated in the standard manner. That is, when an output intrinsic writes a logical record, the record is physically transferred to the file only when the entire physical record (block) of which it is a part is transferred. (Calls to FWRITE, FUPDATE, and FREMOVE send output to the KSAM file.)

For example, the following intrinsic call activates critical output verification:

```
FSETMODE(FILNUM,2);
```

If you want to return to normal output mode, you can use the call:

```
FSETMODE(FILNUM,0);
```

When the file is first opened and when it is opened subsequently following an FCLOSE call, the critical output verification mode is deactivated.

FSPACE

INTRINSIC NUMBER 5

Spaces forward or backward on a file.

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

The FPSACE intrinsic allows you to space forward or backward a specified number of records on a KSAM file. The logical record pointer is repositioned by FSPACE in key sequence. The spacing is based on primary key sequence unless an alternate key has been specified in a prior call to FFINDN, FFINDBYKEY, or FREADBYKEY.

PARAMETERS

filenum *integer by value (required)*

A word identifier supplying the file number of the file on which spacing is to be done.

displacement *integer by value (required)*

An integer indicating the number of logical records to be spaced over, relative to the current logical record pointer position. Record sequence for spacing is based on key sequence. A positive value signifies forward spacing, a negative value signifies backward spacing; zero signifies no spacing, but sets a flag so that the next call to FREAD does not move the logical record pointer before reading the record. The maximum positive value is 32767; the maximum negative value is −32768. The sign is optional for positive values.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FSPACE

If you want to space back a particular number of records in key sequence, you would specify a negative value for the *displacement* parameter in a call to FSPACE. To space forward, you would use a positive or unsigned integer as the *displacement* value. In either case, the displacement indicates the number of records to space over.

For example, suppose the following sequence of primary key values:

```
ABLE
```

```
BAKER <---------- (2) pointer after FSPACE(FILNUM,-2);

CHARLIE

DOG <------------ (1) current record pointer

EASY

FOX <------------ (3) pointer after FSPACE(FILNUM,4);
```

Suppose the current record pointer is at the beginning of the record whose primary key contains the value DOG. To position the pointer to the beginning of the record with a primary key value BAKER:

```
FSPACE(FILNUM,-2);
```

To space forward from the beginning of the record with BAKER as the key value to the beginning of the record with FOX as the key value:

```
FSPACE(FILNUM,4);
```

Figure 4-12 shows that the movement of the pointer bears no relation to the physical placement of records in the file.

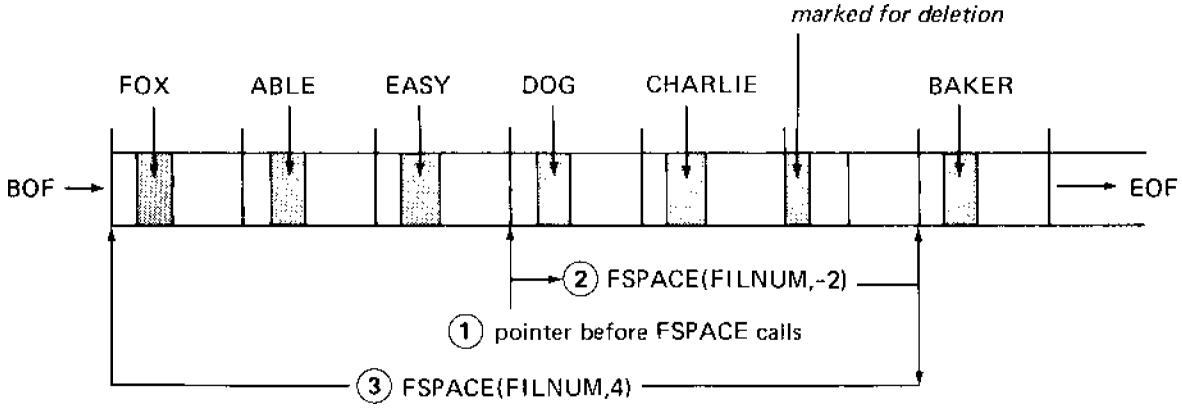


Figure 4-12. File Position with FSPACE

POINTER POSITION

FSPACE checks a flag to determine whether to advance the pointer *before* it moves the pointer the specified number of records. If FSPACE follows a call that reads the file (FREAD or FREADBYKEY) then it advances the pointer to the record in key sequence following the

record just read. After advancing the pointer, FSPACE positions the pointer as indicated in the call. If, on the other hand, FSPACE follows FPOINT, FFINDBYKEY, or FFINDN, the pointer remains positioned to the record specified in one of these calls until FSPACE is executed.

To illustrate, consider the following calls:

```
FREAD <------------- read record, set flag to advance pointer
FSPACE(-1) <--------------- test flag, advance pointer, then move pointer back 1 record
FREAD <----------------- reread record just read
```

SHARED ACCESS

Because FSPACE is a pointer-dependent procedure (see Table 4-2), it is essential to lock the file before the call that determines the original pointer position, then call FSPACE, then call any other procedures that depend on where FSPACE positioned the pointer. When all the pointer-dependent procedures are complete, then unlock the file for other users. To illustrate:

```
FLOCK <-------------- lock file
FFINDBYKEY <----------------- locate a particular key ualue
FSPACE <------------- move pointer relative to that key position
FREAD <-------------- read the record to which pointer is positioned
FUNLOCK <-------------- unlock the file
```

FUNLOCK

INTRINSIC NUMBER 16

Dynamically unlocks a KSAM file.

```
                    IV
FUNLOCK(filenum);
```

The FUNLOCK intrinsic dynamically unlocks a KSAM file (Resource Identification Number) that has been locked with the FLOCK intrinsic.

PARAMETERS

filenum *integer by value (required)*

A word supplying the file number of the file to be unlocked.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FUNLOCK

A file that has been locked with FLOCK in order to allow exclusive updating should be unlocked with FUNLOCK as soon as the update is complete. Dynamic locking and unlocking apply to files opened for this capability. In the *aoptions* parameter of FOPEN, bit 10 must be set to 1 in order to use either FLOCK or FUNLOCK. (For more discussion of dynamic locking and unlocking, refer to the FLOCK intrinsic description.)

Suppose a file has been locked to allow update of a record. To unlock the file following completion of the update, use the call:

```
FUNLOCK(FILNUM);
```

When FUNLOCK is executed, all output written while the file was locked is transferred to the file so that other users have the most recent data.

FUPDATE

INTRINSIC NUMBER 4

Updates the contents of a logical record in a KSAM file.

```
              IV        LA      IV
FUPDATE(filenum,target,tcount);
```

The FUPDATE intrinsic can be used to update a logical record in a KSAM file. The entire record including primary and any alternate keys can be updated with FUPDATE. The record to be updated is the record last referenced by the intrinsics FREAD, FREADBYKEY, FFINDBYKEY, or FPOINT. The new values for the record are moved from the user's stack into this record. The file containing this record must have been opened with the *aoption* parameter of FOPEN set to update access. FUPDATE can be used to update both fixed-length and variable-length records. FUPDATE can be used to modify key values or to change record size, but if key values or the record size are changed, the update operation causes the entire record to be deleted and then rewritten. After an update, a subsequent call to FREAD will read the next record in ascending key sequence after the record just written.

FUPDATE checks only the logical record pointer, not the chronological pointer, in order to determine which record to update. Therefore, if you want to update a record based on its chronological position, precede the call to FUPDATE by a call to FPOINT. FPOINT locates the record by its record number and sets the logical, as well as the chronological, pointer. If you try to locate a record for FUPDATE by calling FREADDIR or FREADC, which only set the chronological pointer, the wrong record will be updated.

If the file was opened for shared access (*aoptions* bits 8,9 = 11), then you must call FLOCK to lock the, file before calling FUPDATE. Note that the file must also have been opened with dynamic locking allowed (*aoptions* bit 10 = 1).

PARAMETERS

filenum	*integer by value (required)* A word identifier supplying the file number of the file to be updated.
target	*logical array (required)* Contains the record to be written in the updating.
tcount	*integer by value (required)* An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies words; if it is negative, it signifies bytes. If *tcount* is less than the *recsize* parameter associated with the record, only the first *tcount* bytes or words are written.

CONDITION CODES

CCE	Request granted.
CCG	An end-of-file condition was encountered during updating.
CCL	Request denied because of an error, such as *tcount* exceeds the record size defined for the KSAM file; or *tcount* does not include all the keys; or a disc input/output error.

SPECIAL CONSIDERATIONS

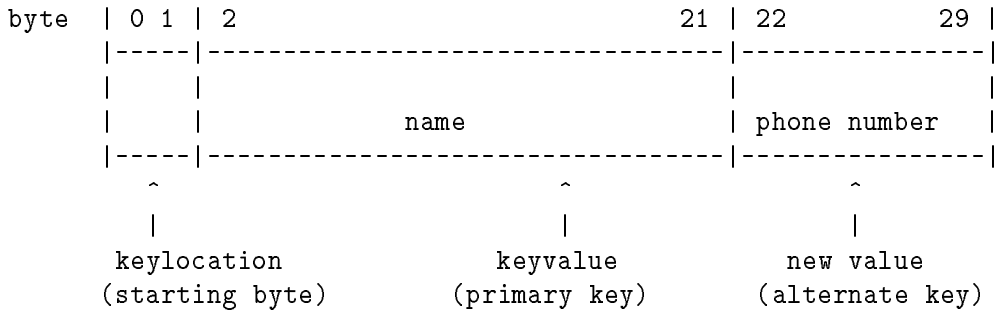
Split stack calls permitted.

USING FUPDATE

In order to update a record in a KSAM file, you must open the file for update. This access mode is specified by setting bits 12 through 15 of the FOPEN *aoptions* parameter to the octal value 5 (binary value 0101). You must then access the record to be updated. Normally, you would read the record with one of the read intrinsics and then modify the record just read.

The record to be updated by FUPDATE is the last record accessed. FUPDATE writes the contents of a user buffer area (*target*) over the existing contents of the last record accessed. The record written by FUPDATE must contain all the key values expected by the file. If only a portion of the record is specified by a *tcount* parameter less than the original record size, then this portion must contain all primary and alternate key values or a CCL condition is returned and the update does not take place.

The example in Figure 4-13 shows an update of an alternate key, the telephone number located in bytes 21 through 28 of the record. In order to locate the record to be updated, FREADBYKEY is executed before FUPDATE. The data input through the standard input device contains the *keylocation* and *keyvalue* values for FREADBYKEY as well as the new value for the update:



Note that bytes are numbered from zero in the standard input or output device, but bytes in the KSAM record are numbered starting from 1 for the *keylocation* parameter.

SHARED ACCESS.

When access is shared, it is essential to lock the file with a call to FLOCK before rewriting any records. After the update, you should unlock the file with FUNLOCK. To make sure you are updating the correct record, include both the intrinsic that locates the record and FUPDATE between the same pair of FLOCK and FUNLOCK intrinsics.

For example, suppose you use FREADBYKEY to examine the record to be updated, you should lock the file before calling the intrinsic that locates the record to be updated and unlock if after the update:

```
FLOCK
FREADBYKEY (or FFINDBYKEY) <----------- locate record to be updated
⋮
FUPDATE <------- update record
FUNLOCK <----------- all key buffers, data buffers and control information written to disc
```

If you perform operations on a record between locating it and updating it, and you do not want to lock the file during this process (between the read and the update), then you can use the following code sequence:

```
FLOCK
FREADBYKEY (or FFINDBYKEY) <-----  locate record
FUNLOCK
.
. <-----------------  while you decide whether to update record,
. other users can modify or delete it
(decide to update)
FLOCK
FREADBYKEY (or FFINDBYKEY) <-----------  reread record
FUPDATE
FUNLOCK
```

UPDATING RECORDS WITH DUPLICATE KEYS.

If you want to sequentially update all the records in a chain of records with duplicate keys, locate the first record in the chain with FFINDBYKEY, FREADBYKEY, or FPOINT. Then call FUPDATE to modify this record. If no key value (the selected key or any other) is modified, subsequent calls to FUPDATE will modify the next sequential records in the chain of records with 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 update all records with duplicate keys, precede each call to FUPDATE with a call to FFINDBYKEY, FREADBYKEY, or FPOINT to position to the beginning of the chain.

If you are in the middle of a duplicate key chain and FUPDATE modifies a key value, you can position back to the next duplicate key in the chain with the following sequence of calls:

```
FSPACE(FILNUM,1); <--------  position to next sequential record
FGETINFO(FILNUM,,,,,,,,RECPTR); <---------------  retrieve current record number
FSPACE(FILNUM,-1); <---------------  backspace to current record
FUPDATE(FILNUM,OUTPUT,-72); <------------  modify key, positioning to end of key chain
.
.
.
FPOINT(FILNUM,RECPTR); <--------  position to next duplicate key using record number
                                  retrieved by FGETINFO
```

Note that if the KSAM file has fixed-length records or if the updated record is the same size as the old record, the space in the data file is reused. Otherwise, the updated record is written to the end of the data file.

```
$CONTROL MAIN=JEXAMPL8
<<*****************************************************>>
<<*                                                   *>>
<<*                     EXAMPLE 8                     *>>
<<*        UPDATE A RECORD IN A KSAM FILE             *>>
<<*                                                   *>>
<<*****************************************************>>
INTEGER FILUM;
INTEGER ERRORCODE;
INTEGER LENGTH;
BYTE ARRAY FILNAME (0:9);="JEXAMPLE ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
BYTE ARRAY OUTPUTB(*)=OUTPUT;
BYTE ARRAY INFO(0:35);
ARRAY INFOW(*)=INFO;
BYTE ARRAY KEYVALUE(*)=INFO(2);
INTEGER KEYLOCATION;
INTRINSIC FOPEN,FCLOSE,FUPDATE,FREADBYKEY,READ,PRINT,
          BINARY,FCHECK,FERRMSG,TERMINATE;
<<************************>>
<<* OPEN THF KSAM FILE   *>>
<<************************>>
FILNUM:=FOPEN(FILNAME,3,5); <<OPEN THE KSAM FILE FOR UPDATE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK (FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*********************************************>>
<<* READ IN KEYVALUE AND KEYLOCATION INFOMATION *>>
<<*********************************************>>
L1:
READ(INFOW,-36);
```

Figure 4-13. FUPDATE Example

```
IF >
THEN BEGIN
       FCLOSE(FILNUM,0,0,); <<CLOSE THE KSAM FILE>>
       IF <> THEN
          BEGIN
            MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE";
            PRINT (MESSAGE,-26,0);
            FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
            FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
            PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
           END
        TERMINATE;
      END;
IF <
THEN BEGIN
        MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
                  PRINT(MESSAGE,-34,0);
                  TERMINATE;
               END;
           PRINT(INFOW,-36,0); <<TEST READ>>
           KEYLOCATION:=BINARY(INFO,2); <<CONVERT FROM ASCII TO BINARY>>
           <<*****************************************************>>
<<* READ KSAM ACCORDING TO KEYVALUE AND KEYLOCATION   *>>
<<*****************************************************>>
           FREADBYKEY(FTLNUM,INPUT,-72,KEYVALUE,KEYLOCATION);
           IF <>
           THEN BEGIN <<ERROR OCCUPRED IN FREADBYKEY>>
                  MOVE MESSAGE:="ERROR OCCUPRED IN FREADBYKEY";
                  PRINT(MESSAGE,-28,0);
                  FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
                  FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
                  PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
                  GO TO L1;
               END;
           <<*********************************************>>
<<*       UPDATE THE RECORD JUST READ          *>>
<<*********************************************>>
           MOVE OUTPUTB(20):=INFO(22),(8);
           FUPUATE(FILNUM,OUTPUT,-72);
           IF <>
           THEN BEGIN
                  MOVE MESSAGE:="ERROR OCCURRED DURING UPDATE";
                  PRINT(MESSAGE,-28,0);
                  FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
                  FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
                  PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
                  TERMINATE;
               END;
```

Figure 4-13. FUPDATE Example (continued)

```
        <<*********************************************>>
        <<* PRINT THE RECORD JUST UPDATED *>>
        <<*********************************************>>
        PRINT(OUTPUT,-72,0);
        <<*********************************>>
        <<* GO BACK TO GET ANOTHER RECORD *>>
        <<*********************************>>
        GO TO L1;
        END;

        Output from Program Execution:


              read from $STDIN
             /
            /                        / updated record
           /                        /
          /                        /
01WHITE GORDON 428-2498           /
WHITE GORDON 428-2498 4350 ASHBY AVE. BERKELEY CA. 91234
01ECKSTEIN LEO 263-2464
ECKSTEIN LEO 263-2464 5303 STEVENS CREEK SANTA CLARA CA. 95050
```

Figure 4-13. FUPDATE Example (continued)

FWRITE

INTRINSIC NUMBER 3

Write a logical record from the user's stack to a KSAM file.

```
                IV       LA     IV      LV
FWRITE(filennum,target,tcount,control);
```

The FWRITE intrinsic writes a logical record from the user's stack to the KSAM file. The record contents are contained in the array *target* in the user's program and include all key values. FWRITE uses the primary key value to update the key file so that the new record is in sequence by primary key value. Any alternate keys are also entered into their appropriate positions in the key file. No separate key specification is required since all the key values are contained in the record to be written.

Following execution of FWRITE, the logical record pointer is positioned at the next sequential record in key sequence or at the end-of-file if the record is the last in sequence. The particular key is the current key being used when FWRITE is called.

If sequential processing was specified for the file in the flagword of *ksamparam* when the file was opened by FOPEN, then the records must be written in ascending order by primary key. If duplicate keys are not allowed, any record with a key duplicating a key in an existing record is not written and a CCL condition code is returned.

When the physical bounds of either the data file or the key file prevent further writing (all allowable extents are filled), an end-of-file condition code (CCG) is returned to the user's program.

If the file was opened for shared access (*aoptions* bits 8,9 = 11), then you must dynamically lock the file with FLOCK before calling FWRITE. Note that the file must also have been opened for dynamic locking (*aoptions* bit 10 = 1).

PARAMETERS

filenum — *integer by value (required)*

A word identifier supplying the file number of the file to be written on.

target — *logical array (required)*

Contains the record to be written.

tcount — *integer by value (required)*

An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If *tcount* is less than the *recsize* parameter associated with the record, only the first *tcount* words or bytes are written.

If *tcount* is larger than the *recsize* value, the write request is refused and condition code CCL is returned.

control — *logical by value (required)*

A logical value representing a carriage control code. This parameter has no meaning for KSAM files but must be included for compatibility. Whatever value is specified will be ignored.

CONDITION CODES

CCE	Request granted.
CCG	The physical bounds of the file prevented further writing; all disc extents are filled.
CCL	Request denied because an error occurred, such as: an input/output error occurred; a duplicate key value occurred when duplicates are not allowed; *tcount* does not include all keys; or sequential processing was specified in the flagword of *ksamparam* in FOPEN and the primary key is not the next in ascending order.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FWRITE

The FWRITE intrinsic writes records from an array in your program to a KSAM file. All the key information is contained in this *target* array. The record is written to the data file and the keyfile is updated to reflect the primary key and any alternate keys in the new record.

Depending on how the file was opened, you can write records at random regardless of primary key order, or you may be constrained to write records in sequential order by primary key value. The examples in this manual use the file JEXAMFIL that is created for writing at random. If you refer to Figure 4-5, the flagword of the *ksamparam* parameter is set to the binary value 0000000000000010. Bit 14, indicating that record numbers start with 1, is the only bit set. If bit 13 had also been set to 1 then all records written to the file would have to be in ascending order by primary key value. In such a case, the chronological order of records and the sequential order would be the same.

When you write a record to a KSAM file, FWRITE either overwrites any records previously written to the file or else writes new records following existing records. The choice is made when you open the file. *If you set bits 12 through 15 of the aoptions parameter of FOPEN to the binary value 0001 (octal or decimal 1), then all records written to the file before this open are deleted and FWRITE writes records to a cleared file.* If you set bits 12 through 15 of *aoptions* to 0010 or 0011 (octal or decimal 2 or 3), then any previously written data is saved. The example in Figure 4-14 deletes any data written to file JEXAMFIL before it was opened. The file will have no data other than that written by this program. If, after closing the file, you want to open it to write more data without deleting existing data, then you must set the *aoptions* access type (bits 12-15) to 0010 or 0011.

SHARED ACCESS

When access is shared, it is essential that you lock the file before writing new records. This means opening the file with dynamic locking allowed and calling FLOCK before calling FWRITE. You should also unlock the file with FUNLOCK after writing the records.

```
$CONTROL MAIN=EXAMPLE9
<<*****************************************************>>
<<*                                                   *>>
<<*                     EXAMPLE  9                    *>>
<<*           WRITE TO EXISTING KSAM FILE             *>>
<<*                                                   *>>
<<*****************************************************>>
INTEGER FILNUM;
INTEGER ERRORCODE;
INTEGER LENGTH;
BYTE ARRAY FILNAME(0:9):="JEXAMFIL ";
ARRAY MESSAGE(0:35);
ARRAY INPUT(0:39);
ARRAY OUTPUT(*)=INPUT;
INTRINSIC FOPEN,FCLOSE,FWRITE,READ,PRINT,FCHECK,FERRMSG;
INTRINSIC TERMINATE;
<<************************>>
<<* OPEN THE KSAM FILE *>>
<************************>>
FILNUM:=FOPEN(FILNAME,3,2); <<OPEN FILE FOR WRITE>>
IF FILNUM=0
THEN BEGIN <<CANNOT OPEN KSAM FILE>>
       MOVE MESSAGE:="CANNOT OPEN KSAM FILE";
       PRINT(MESSAGE,-21,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*********************************>>
<<* READ DATA FROM $STDIN DEVICE *>>
<<*********************************>>
L1;
READ(INPUT,-72); <<READ ONE RECORD FROM $STDIN>>
IF >
THEN BEGIN <<END OF FILE ON $STDIN>>
       FCLOSE(FILNUM,1,0); <<CLOSE THE KSAM FILE>>
       IF <> THEN
         BEGIN <<CANNOT CLOSE THE KSAM FILE>>
           MOVE MESSAGE:="CANNOT CLOSE THE KSAM FILE";
           PRINT(MESSAGE,-29,0);
           FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMRER>>
           FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
           PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
         END;
       TERMINATE;
     END;
```

Figure 4-14. FWRITE Example

```
IF <
THEN BEGIN
       MOVE MESSAGE:="ERROR OCCURRED WHILE READING INPUT";
       PRINT(MESSAGE,-34,0);
       TERMINATE;
     END;
PRINT(OUTPUT,-72,0); <<ECHO CHECK>>
<<*********************************************>>
<<* WRITE THE DATA JUST READ TO THE KSAM FILE *>>
<<*********************************************>>
FWRITE(FILNUM,OUTPUT,-72,0);
IF <>
THEN BEGIN <<ERROR OCCURRED WHILE WRITING KSAM>>
       MOVE MESSAGE:="ERROR OCCURRED WHILE WRITING KSAM FILE";
       PRINT(MESSAGE,-38,0);
       FCHECK(FILNUM,ERRORCODE); <<GET ERROR NUMBER>>
       FERRMSG(ERRORCODE,MESSAGE,LENGTH);<<CONVERT TO STRING>>
       PRINT(MESSAGE,-LENGTH,0); <<PRINTOUT ERROR MESSAGE>>
       TERMINATE;
     END;
<<*********************************>>
<<* GO BACK TO GET ANOTHER RECORD *>>
<<*********************************>>
GO TO L1;
END;
```

Figure 4-14. FWRITE Example (continued)

FWRITEDIR

INTRINSIC NUMBER 8

Writes a specific logical record from the user's stack to a disc file.

Note This intrinsic may not be used for KSAM files. If called for a file created as a KSAM file, the intrinsic returns a CCL condition code.

FWRITELABEL

INTRINSIC NUMBER 20

Writes a user file label.

```
                    IV      LA      IV      IV      O-V
FWRITELABEL(filenum,target,tcount,labelid);
```

The FWRITE LABEL intrinsic writes a user-defined label onto a disc file. This intrinsic overwrites old user labels.

PARAMETERS

filenum *integer by value (required)*

A word identifier specifying the file number of the file to which the label is to be written.

target *logical array (required)*

Contains the label to be written to the disc file.

tcount *integer by value (optional)* An integer specifying the number of words to be transferred from the array.

Default: 128 words.

labelid *integer by value (optional)* An integer specifying the number of the label to be written. The first label is 0.

Default: A default value of 0 is assigned.

CONDITION CODES

CCE	Request granted.
CCG	Request denied because the calling process attempted to write a label beyond the limit specified in FOPEN when the file was opened.
CCL	Request denied because an error occurred.

SPECIAL CONSIDERATIONS

Split stack calls permitted.

USING FWRITELABEL

You can write your own labels to a KSAM file with the FWRITELABEL intrinsic. Such labels are useful to hold information related to the file but not part of it. For example, you might use a label to contain the date and time of the last update to the file.

The number of labels that are allowed to be written to any file must be specified in the *userlabels* parameter of the FOPEN intrinsic when the file was created. If an attempt is made to write more labels than are specified for the file at creation, a CCG condition is returned.

In order to write labels, as with any other write request, the file must be opened for write access. This means that the *aoptions* parameter of FOPEN must be set to one of the following:

```
bits 12:4 = 0001 (octal 1) \
            0010 (octal 2) |- write only access
            0011 (octal 3) /
          = 0100 (octal 4) <------ input/output access
          = 0101 (octal 5) <------ update access
```

Suppose file KDATA has been created as follows:

```
KFILNUM:=FOPEN(KDATA,%4004,4,,,,2);
                          /  |    |
                         /   |    |
                        /    |    |
          new KSAM file,     |   number of labels
          ASCII coded        |
                             input/output access
```

Then a total of two labels, each with a maximum of 128 words, can be written to this file with FWRITELABEL. To write a second label consisting of 60 words stored in the array LABELBUF, use the following call:

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

Note that label numbering starts with zero, so the second label is identified by the number 1.

HP32208

INTRINSIC NUMBER 308

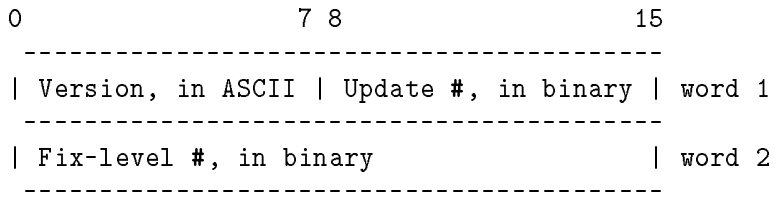
Returns current version, update, and fix level of KSAM/3000.

```
          D
version:=HP32208
```

The double word result returned by HP32208 contains the version number in ASCII, the update number in binary, and the fix-level number in binary of the KSAM/3000 version currently in use.

FUNCTIONAL RETURN

version Double word returned by HP32208 in the form:



CONDITION CODES

Condition codes are not affected by execution of this intrinsic.

USING HP32208

You may call this intrinsic in order to get the current version, update, and fix numbers of the KSAM/3000 that is currently being used. The intrinsic FGETKEYINFO returns the version, update, and fix number of a KSAM file at the time the file is created (refer to words 16/17 of the *ksamcontrol* parameter, Table 4-4). The version, update, and fix number of a KSAM file at creation is also returned by the VERIFY command (refer to section II). You can call HP32208 to get the KSAM version you are using in order to compare it with the version at file creation of a file you are accessing.

Another reason for calling HP32208 is if you want to convert the version, update, and fix numbers to display values so they can be listed for documentation purposes.

5

USING KSAM FILES IN FORTRAN PROGRAMS

OVERVIEW

The FORTRAN language has no input/output statements that can be used to access or create a KSAM file directly. In order to reference KSAM files for input or output, the FORTRAN progammer can choose between using MPE file system intrinsics (as described in section IV) or using COBOL procedures (described in section III). He can create a KSAM file with the KSAMUTIL utility program (described in section II) or with a call to the FOPEN intrinsic, but not with a COBOL procedure.

If you are programming in FORTRAN, you can use the CALL statement to call any of the COBOL procedures or any of the file system intrinsics that access KSAM files. In order to determine which to use, you should refer to Table 3-1 for a list of the COBOL procedures that provide KSAM interface and to Table 4-1 for a similar list of the file system intrinsics used for KSAM interface. You will note that there are differences in the functions provided.

Since the COBOL procedures are described in detail in section III and the file system intrinsics are described in detail in section IV, these descriptions are not repeated here. This section merely describes how to call the COBOL procedures or the file system intrinsics, and provides examples of file creation and access along with brief commentaries.

CALLING FILE SYSTEM INTRINSICS

To the FORTRAN user, some of the file system intrinsics are treated as functions and others as subroutines. A function is called *implicitly* by being referenced in a FORTRAN statement. A subroutine is called *explicitly* with the FORTRAN CALL statement. A further distinction is that a function can return a value to the calling program as a functional return, whereas a subroutine can return values only through the parameters (arguments) specified in the call.

To illustrate, the FOPEN intrinsic is called as a function:

```
FILNUM=FOPEN(FILENAME,%4004L,%101L,-72,DEVICE,KSAMPARAM,,10,,100J)
```

When this statement is executed, a value is returned to the integer variable FILNUM. Note that the word CALL is not used. On the other hand, the FWRITE intrinsic is a subroutine that must be called with the CALL statement:

```
CALL FWRITE(FILNUM,OUTPUT,-72,%0L)
```

In order to determine quickly which is which, look up the intrinsic definition in section III; if it has a functional return it should be called as a function, if not it should be called as a subroutine.

MPE/3000 system intrinsics differ from FORTRAN/3000 language procedures: System intrinsics can have optional parameters (arguments) whereas all parameters must be specified in a call to a FORTRAN procedure. Another difference is that parameters can be passed by value to a system intrinsic but they must be passed by reference to a FORTRAN procedure. To pass a parameter by value, use the literal value as a parameter (the parameter −72 in the FOPEN call above); to pass by reference, the value is assigned to a parameter specified as a variable or array name (FILENAME in the FOPEN call).

In order to take advantage of the capabilities of the system intrinsics, you should declare the names of any intrinsics you plan to use in a SYSTEM INTRINSIC statement. This statement must appear as one of the declaration statements that precede executable statements in a FORTRAN/3000 program. For example, if you plan to call FOPEN, FCLOSE, FWRITE, and FCHECK then these intrinsics should be declared in the statement:

```
SYSTEM INTRINSIC FOPEN,FCLOSE,FWRITE,FCHECK
```

Declared in this way, you can then omit optional parameters from the call and pass parameters by value. If you do not declare the intrinsics in a SYSTEM INTRINSIC call, then a function call such as that illustrated above for FOPEN would generate an error because it omits some parameters and passes others by value.

CALLING COBOL PROCEDURES

Like the FORTRAN/3000 procedures, COBOL/3000 procedures do not allow you to omit any parameters from the parameter list or to pass parameters by value. Thus no special provisions need be made in order to call COBOL procedures from a FORTRAN program. Since the COBOL procedure call differs in format from the FORTRAN procedure call, you must translate from the COBOL format when calling a COBOL procedure in a FORTRAN program. The translation is simple:

```
CALL "CKOPEN" USING filetable, status. <-------------- COBOL format
       |             |   |----|
       |             |   |
    procedure    parameters
      name        |      |
       |          |      |
       |          |      |
CALL CKOPEN (filetable, status) <---------------------- FORTRAN format
```

CREATING A KSAM FILE WITH A CALL TO FOPEN

A KSAM file can be created with the >BUILD command of the KSAMUTIL program or it can be created programmatically through a call to the file system intrinsic FOPEN. Figure 5-1 contains a FORTRAN program that uses the intrinsic FOPEN to create and open a Ele, and the intrinsic FWRITE to write to the open file. It checks for errors with the FCHECK and FERRMSG intrinsics, and closes the file with a call to FCLOSE.

The file is named FEXAMFIL and the associated key file is named FKEYFILE. Two keys are used, a primary key of 20 characters starting in byte 1 of each data record, and an alternate key of eight characters starting in byte 21 of the data record. The primary key contains a name, the alternate key a phone number (refer to the input data in Figure 5-1).

DEFINING KSAMPARAM

The parameter *ksamparam* describes the key file in an array that contains many different types of data (refer to Table 4-7). Because the data differs, the EQUIVALENCE statement is used to equate the word-array KSAMPARAMA to the byte-array KSAMPARAM to the double-word-array KSAMPARAMD. The keyfile name is in the first eight bytes and this is equivalenced to the beginning of the array. The key device is defined in word 7 of KSAMPARAMA, and the key descriptions begin in word 18.

The flag word (word 17) has the octal value 2. This means that only bit 14 is set to 1. The flagword defines the following options for the KSAM file:

bit 13 = 0 file is permanently saved in system directory

bit 14 = 1 record numbers in file start with 1, not zero

bit 15 = 0 records can be written in random order

If you compare this *ksamparam* definition to that in the SPL sample program (Figure 4-5), you will note that the index values into the array differ. This is because, SPL arrays begin numbering with zero whereas FORTRAN arrays begin numbering with one.

CALLING FOPEN

In the FOPEN call, the first parameter is the KSAM file name that identifies the data file and the KSAM file as a whole. The second parameter specifies the file options (*foptions*) parameter as octal 4004:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	binary
0	0			4			0	0		0			4			octal

This defines the following file options:

New KSAM file	(bit 4=1)
Allow :FILE	(bit 5=0)
Fixed-Length Records	(bits 8,9=00)
ASCII code	(bit 13=1)
New file	(bits 14,15=00)

```
C*********************************************************************
C                                                                    *
C                          EXAMPLE 1 *                               *
C                      BUILD A KSAM FILE *                           *
C                                                                    *
*********************************************************************
          SYSTEM INTRINSIC FOPEN,FCLOSE,FWRITE,FERRMSG,FCHECK
          INTEGER KSAMPARAMA(26)
          INTEGER KEYDESCRIPTION(8)
          CHARACTER KSAMPARAM(52)
          INTEGER*4 KSAMPARAMD(13)
          CHARACTER*8 KEYFILENAME
          CHARACTER*17 KEYDEVICEE
          EQUIVALENCE ((KSAMPARAMA,KSAMPARAM,KSAMPARAMD,KEYFILENAME)
          EQUIVALENCE (KEYDEVICE,KSAMPARAMA(7))
          EQUIVALENCE (KSAMPARAMA(18),KEYDESCRIPTION)
          INTEGER FILNUM
          INTEGER LENGTH
          CHARACTER FILENAME*10
          CHARACTER DEVICE*10
          CHARACTER*72 INPUT
          LOGICAL OUTPUT(36)
          CHARACTER MESSAGE(72)
          EQUIVALENCE MESSAGEW(36)
          EQUIVALENCE (MESSAGE,MESSAGEW)
          EQUIVALENCE (INPUT,OUTPUT)
          DATA FILENAME/"FEXAMFIL  "/ <---------- filename
          DATA DEVICE/"DISC  "/ device
          DATA KEYDEVICE/"DISC  "/ device
          DATA KEYFILENAME/"FKEYFILE"
          DATA KSAMPARAMD(3)/100J/ <-------------- file size
          DATA KSAMPARAMA(16)/2/ <---------------- flagword
          DATA KSAMPARAMA(17)/2/ <---------------------- no. of keys
          DATA KEYDESCRIPTION/%{4/1,12/20], 1,%[1/0,15/4],0,    key
         1                    %[4/1,12/ 8],21,%[1/0,15/4], 0/ descriptions
C*********************************************************************
C                                                                    *
C                      OPEN THE KSAM FILE                            *
C                                                                    *
C*********************************************************************
         FILNUM=FOPEN(FILENAME,*4004L,%101L,-72,DEVICE,KSAMPARAM,M,
        1              ,10,,100J)
         IF (FILNUM .EQ. 0)  GO TO 400
```

Figure 5-1. Creating and Writing to KSAM File in FORTRAN

```
*************************************************************************
C                                                                       *
C                      READ DATA FROM $STDIN                            *
C                                                                       *
*************************************************************************
         20 READ (5,300,END=30,ERR=40)   INPUT
*************************************************************************
C                                                                       *
C           WRITE THE DATA JUST READ TO THE KSAM FILE                   *
C                                                                       *
C*************************************************************************
          50 DISPLAY INPUT
                   CALL FWRITE(FILNUM,OUTPUT,-72,%OL)
                   IF (.CC.) 70,20,70
```

Creating and Writing to KSAM File in FORTRAN

```
 *************************************************************************
C                                                                       *
C                            ERROR MESSAGE                              *
C                                                                       *
C*************************************************************************

          70 STOP "ERROR OCCURRED WHILE WRITING KSAM FILE"
         100 STOP  "END OF JOB"
          30 CALL FCLOSE(FILNUM,0,0)
             IF (.CC.) 33,100,33
          33 STOP "CAN NOT CLOSE THE KSAM FILE"
          40 STOP "ERROR OCCURRED WHILE READING INPUT"
         400 CALL FCHECK (FILNUM,IERRNUM)
             CALL FERRMSG (IERRNUM,MESSAGEW,LENGTH)
             WRITE(6,200) (MESSAGE(I),I=1,LENGTH)
             STOP "CAN NOT OPEN KSAM FILE"
         300 FORMAT(A72)
         200 FORMAT(1X,72A1)
             END

             Output from Program Execution:

          NOLAN      JACK    923-4975  967 REED AVE.           SUNNYVALE
          HOSODA     JOE     227-8214  1180 SAINT PETER CT.    LOS ALTOS
          ECKSTEIN   LEO     287-5137  5303 STEVENS CREEK      SANTA CLARA
          CARDIN     RICK    578-7018  11100 WOLFE ROAD        CUPERTINO
          PASBY      LINDA   295-1187  TOWN & CNTRY VILLAGE   SAN JOSE
          SEELY      HENRY   293-4220  1144 LEBERTY ST.        EL CERRITO
          ROBERT     GERRY   259-5535  12345 TELEGRAPH AVE.    BERLELEY
          TURNEWR    IVAN    984-8498  22905 EMERSON ST.       OAKLAND
          WHITE      GORDON  398-0301  4350 ASHBY AVE.         BERKELEY
          WESTEP     ELDER   387-4598  1256 KINGFISHER ST.     SUNNYVALE
STOP END OF JOP
```

The next parameter defines the access options (*aoptions*) as the octal value 101:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	binary
0		0			0			1			0			1		octal

This defines the following access options:

KSAM access expected	bit 3=0)
Exclusive access	bits 8,9=01)
No dynamic locking	bit 10=0)
Write only access	bits 12-15=0001)

A new file contains no information and is always opened for write access. Before accessing the file for reading or update, it must be reopened. Such an open specifies that the file is an old file in the *foptions* parameter. Depending on the type of access expected, *aoptions* can be omitted or can specify a particular access type.

CREATING A KSAM FILE WITH KSAMUTIL

Instead of using the file system intrinsic FOPEN to create the KSAM file, you can create the file with the >BUILD command of the KSAMUTIL program. Once created, the file can be opened with a call to FOPEN or CKOPEN. (Note that CKOPEN cannot be used to create a file.)

The same file created in Figure 5-1 with FOPEN could be created in KSAMUTIL as follows:

```
:RUN KSAMUTIL.PUB.SYS
>BUILD FEXAMFIL;REC=-72,10,F,ASCII;DEV=DISC;DISC=100;&
       KEYFILE =FKEYFILE;KEY=B,1,20;KEY=B,21,8;FIRSTREC=1
```

OPENING A KSAM FILE WITH A COBOL PROCEDURE

The CKOPEN procedure requires two parameters: one is a table that identifies the file and specifies the type of access; the other is a two-byte item to which the status of the call is returned. When calling this procedure from a FORTRAN program, the *filetable* parameter must be defined as an eight-word array containing both integer and character values.

Any item that is defined as COMPUTATIONAL or COMP in a COBOL program is declared as an INTEGER in a FORTRAN program when it contains four bytes or less. Thus, the following are equivalent:

```
02 FILENUM PIC S9(4) COMP. <---------------------- COBOL description

INTEGER FILENUM <--------------------------------- FORTRAN description
```

Any data items defined with a picture of X in COBOL would be declared as CHARACTER items in FORTRAN. Thus, the following are equivalent:

```
02 FILENAME PIC X(8). <-------------------------- COBOL description
CHARACTER *8 FILENAME <---------------------- FORTRAN description
```

Assuming that file FEXAMFIL has been created by the >BUILD command, the FORTRAN statements in Figure 5-2 open that file for output only and sequential access.

```
      INTEGER FILETABLE(8)
      CHARACTER FILETABLC(16)
      INTEGER FILENUM
      CHARACTER*8 FILENAME
      CHARACTER*2 FSTAT
      INTEGER IOTYPE
      INTEGER AMODE
      INTEGER PREVOP
      EQUIVALENCE (IFSTAT,FSTAT)
      EQUIVALENCE (FILETABLE,FILETABLC,FILENUM)
      EQUIVALENCE (FILETABLC(3),FILENAME)
      EQUIVALENCE (FILETABLE(6),IOTYPE)
      EQUIVALENCE (FILETABLE(7),AMODE)
      EQUIVALENCE (FILETABLE(8),PREVOP)
      DATA FILENAME/"FEXAMFIL"/,PREVOP/0/

C*********************************************************************
C OPEN KSAM FILE FOR SEQUENTIAL INPUT                               *
C*********************************************************************

      IOTYPE=1                    I/O type is output only
      AMODE=0                        access mode is sequential
      CALL CKOPEN(FILETABLE,IFSTAT)
```

Figure 5-2. Opening KSAM File with CKOPEN

WRITING TO A KSAM FILE

Once a KSAM file has been created and opened for output access, you can write to the file with a call to FWRITE or a call to CKWRITE. You may choose to write records in primary key sequence and have that sequence checked. To do this, you can open the file for sequential access with CKOPEN or else call FOPEN with bit 13 of the flagword in *ksamparam* set to 1. If the sequence in which records are written doesn't matter, you can open the file for random access in COBOL or open the file by calling FOPEN with bit 13 of the *ksamparam* flagword cleared to zero.

The example in Figure 5-1 uses FWRITE to write records to the KSAM file in the order in which they are read from the standard input device; they are not written in primary key order.

Since duplicate primary keys are never allowed by the COBOL KSAM procedures, you should use the file system intrinsics if you want to allow duplicate primary keys. Duplicate alternate keys are allowed by both the file system and COBOL if so specified when the file was created.

READING A KSAM FILE IN KEY ORDER

PRIMARY KEY SEQUENCE

Reading a file in primary key order requires no other preparation than to open the file (file system) or to open the file for sequential input (COBOL). In the file system, sequential logical read is the default and the *aoptions* parameter can be omitted from the FOPEN call. In a COBOL procedure, input type and sequential access are indicated by zero values in the appropriate words of the *filetable* table.

Once opened for input, the file system FREAD intrinsic or the COBOL CKREAD procedure can be called to read the file in sequence by primary key.

ALTERNATE KEY SEQUENCE

To read a file in sequence by an alternate key, that alternate key must be specified in a call prior to the call to a read procedure or intrinsic. In COBOL, you would use a call to CKSTART; with the file system intrinsics you would use FFINDBYKEY.

The example in Figure 5-3 illustrates use of the file system intrinsics FREAD and FFINDBYKEY to read a KSAM file in sequence first by primary key and then by alternate key.

RANDOM ORDER

A particular record in the file can be selected for access according to the value of a key field in the record. This can be a primary or alternate key field. In COBOL, a call to CKREADBYKEY reads a record specified by the key value parameters. The file system uses the intrinsic FREADBYKEY for the same purpose. The main difference here is that the file must be opened for random access before calling the COBOL procedure; no distinction is made by the MPE file system between a file opened for sequential access and one opened for random access.

```
C*****************************************************
C                       EXAMPL2                     *
C                  READ KSAM FILE SEQUENTIALLY      *
C*****************************************************
         SYSTEM INTRINSIC FOPEN,FCLOSE,FWRITE,FERRMSG,FCHECK
         SYSTEM INTRINSIC FWREAD,FFINDBYKEY
         INTEGER FILNUM
         CHARACTER FILENAME*10
         CHARACTER OUTPUT*72
         CHARACTER MESSAGE(72)
         LOGICAL INPUT (36)
         LOGICAL MESSAGEW(36)
         EQUIVALENCE (MESSAGE,MESSAGEW)
         EQUIVALLNCE (OUTPUT,INPUT)
         CHARACTER KEYVAL*8
         DATA FILENAME/"FEXAMFIL " /
         DATA KEYVAL/"000-0000"/
C*****************************************************
C             OPEN KSAM FILE FOR INPUT              *
C*****************************************************
         FILNUM=FOPEN(FILENAME,%7L)
           IF (FILNUM .EQ. 0) GO TO 200
C*****************************************************
C               READ DATA FROM FILE IN              *
C                  SEQUENTIAL ORDER                 *
C*****************************************************
            DISPLAY "PRINT RECORDS IN NAME ORDER"
         20 ILEN=FREAD(FILNUM,INPUT,-72)
             IF (.CC.) 300,30,35
         30 DISPLAY OUTPUT
             GO TO 20
C*****************************************************
C      READ IN SEQUENCE BY ALTERNATE KEY            *
C*****************************************************
         35 DISPLAY "PRINT RECORDS IN PHONE # ORDER"
            CALL FFINDBYKEY(FILNUM,KEYVAL,21,8,2)
            IF (.CC.) 400,40,400
         40 ILEN=FREAD(FILNUM,INPUT,-72)
             IF (.CC.) 500,45,500
         45 DISPLAY OUTPUT
             GO TO 40
C*****************************************************
C                   CLOSE FILE                      *
C*****************************************************
         50 CALL FCLOSE(FILNUM,0,0)
             IF (.CC.) 600,55,600
         55 STOP "END OF JOB"
```

Figure 5-3. Reading KSAM File in Key Sequence Using FORTRAN

```
C*************************************************
C                  ERROR MESSAGES                 *
C*************************************************
          200 CALL FCHECK(FILNUM,IERRNUM)
              CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
              WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
              STOP "CANNOT OPEN KSAM FILE"
          300 CALL FCHECK(FILNUM,IERRNUM)
              CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
              WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
```

```
          400 CALL FCHECK(FILNUM,IERRNUM)
              CALL FERRMSG(IERRMUN,MESSAGEW,LENGTH)
              WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
              STOP " ERROR OCCURRED WHILE USING FIFINDBYKEY"
          500 CALL FCHECK(FILNUM,IERRNUM)
              CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
              WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
              STOP "ERROR OCCURRED READING BY ALTERNATE KEY"
          600 CALL FCHECK(FILNUM,IERRNUM)
              CALL FERRMSG(IERRNUM,MESSGEW,LENGTH)
              WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
              STOP "CANNOT CLOSE FILE"
          250 FORMAT(1X,72A1)
               END


Output from Program Execution:

          PRINT RECORDS IN NAME ORDER
          CARDIN     RICK    578-7018  11100 WOLFE ROAD       CUPERTINO
          ECKSTEIN   LEO     287-5137  5303 STEVENS CREEK     SANTA CLARA
          HOSODA     JOE     227-6214  1180 SAINT PETER CT.   LOS ALTOS
          NOLAN      JACK    923-4975  967 REED AVE.          SUNNYVALE
          PASBY      LINDA   295-1187  TOWN & CNTRY VILLAGE   SAN JOSE
          ROBERT     GERRY   259-5535  12345 TELEGRAPH AVE.   BERKELEY
          SEELY      HENRY   293-4220  1144 LEBERTY ST. EL    CERRITO
          TURNEWR    IVAN    984-8498  22905 EMERSON ST.      OAKLAND
          WESTER     ELDER   287-4598  1256 KINGFISHER ST.    SUNNYVALE
          WHITE      GORDON  398-0301  4350 ASHBY AVE.        BERKELEY
          PRINT RECORDS IN PHONE # ORDER
          HOSODA     JOE     227-8214  1180 SAINT PETER CT.   LOS ALTOS
          ROBERT     GERRY   259-5535  12345 TELEGRAPH AVE.   BERKELEY
          WESTER     ELDER   287-4598  1256 KINGFISHER ST.    SUNNYVALE
          ECKSTEIN   LEO     287-5137  5303 STEVENS CREEK     SANTA CLARA
          SEELY      HENRY   293-4220  1144 LEBERTY ST.       EL CERRITO
          PASBY      LINDA   295-1187  TOWN & CNTRY VILLAGE   SAN JOSE
          WHITE      GORDON  398-0301  4350 ASHBY AVE.        BERKELEY
          CAPDIN     RICK    578-7018  11100 WOLFE ROAD       CUPERTINO
          NOLAN      JACK    923-4975  967 REED AVE.          SUNNYVALE
          TURNEWR    IVAN    984-8498  22905 EMERSON ST.      OAKLAND
STOP END OF JOB
```

Figure 5-3. Reading KSAM File in Key Sequence Using FORTRAN (continued)

READING A KSAM FILE IN CHRONOLOGICAL ORDER

The order in which records are physically written to a data file is called chronological order. This order is not necessarily the same as a sequence by key value although it may be. In particular, if the records were written by a COBOL procedure to a file opened for sequential access, then the chronological sequence and the primary key sequence are the same. If, however, these orders differ, then the file system provides an intrinsic that allows you to read a KSAM file in chronological order.

Figure 5-4 is a program that uses the intrinsic FREADC to read the records in the order they were stored in the file.

Other file system intrinsics allow you to position the file to a particular record number in chronological order (FPOINT), to retrieve the current chronological record number (FGETINFO), and to read a record located by its chronological record number (FREADDIR).

The COBOL procedures for KSAM interface do not provide the means to access records by chronological record number.

```
C*************************************************************
C                  EXAMPL3                                  *
C            READ KSAM FILE CHRONOLOGICALLY                 *
C *************************************************************
                SYSTEM INTRINSIC FOPEN,FCLOSE,FERRMSG,FCHECK
                SYSTEM INTRINSIC FREADC
                INTEGER FILNUM
                CHARACTER FILENAME*10
                CHARACTER OUTPUT*72
                CHARACTER MESSAGE(72)
                LOGICAL INPUT (36)
                LOGICAL MESSAGEW(36)
                EQUIVALENCE (MESSAGE,MESSAGEW)
                EQUIVALENCE (OUTPUT,INPUT)
                DATA FILENAME/"FEXAMFIL "/
C*************************************************************
C                   OPEN KSAM FILE FOR INPUT                 *
C*************************************************************
                FILNUM=FOPEN(FILNAME,%7L)
                  IF (FILNUM .EQ. 0) GO TO 200
C*************************************************************
C                     READ DATA FROM FILE IN                 *
C                        CHRONOLOGICAL ORDER                 *
C*************************************************************
                DISPLAY "PRINT RECORDS IN CHRONOLOGICAL ORDER"
           20 ILEN=FREADC(FILNUM,INPUT,-72)
                IF (.CC.) 300,30,50
           30 DISPLAY OUTPUT
                 GO TO 20
C*************************************************************
C                        CLOSE FILE                          *
C*************************************************************
           50 CALL FCLOSE(FILNUM,0,0)
                 IF (.CC.) 600,55,600
           55 STOP "END OF JOB"
```

Figure 5-4. Reading KSAM File in Chronological Sequence Using FORTRAN

```
C*************************************************************
C               ERROR MESSAGES                                *
C*************************************************************
    200 CALL FCHECK(FILNUM,IERRNUM)
        CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
        WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
        STOP "CANNOT OPEN KSAM FILE"
    300 CALL FCHECK(FILNUM,IERRNUM)
        CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
        WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
        STOP "ERROR OCCURRED READING IN CHRONOLOGICAL ORDER"
    600 CALL FCHECK(FILNUM,IERRNUM)
        CALL FERRMSG(IERRNUM,MESSAGEW,LENGTH)
        WRITE(6,250)(MESSAGE(I),I=1,LENGTH)
        STOP "CANNOT CLOSE FILE"
    250 FORMAT(IX,72A1)
        END

Output from Program Execution:

PRINT RECORDS IN CHRONOLOGICAL ORDER
NOLAN      JACK    923-4975  967 REED AVE,           SUNNYVALE
HOSODA     JOE     227-8214  1180 SAINT PETER CT.    LOS ALTOS
ECKSTEIN   LEO     287-5137  5303 STEVE'S CREEK      SANATA CLARA
CARDIN     RICK    578-7018  11100 WOLFE ROAD        CUPERTINO
PASBY      LINDA   295-1187  TOWN & CNTRY VILLAGE    SAN JOSE
SEELY      HENRY   293-4220  1144 LEBERTY ST.        EL CERRITO
ROBERT     GERRY   259-5535  12345 TELEGRAPH AVE.    BERKELEY
TURNEWR    IVAN    984-8498  22905 EMERSON ST.       OAKLAND
WHITE      GORDON  398-0301  4350 ASHBY AVE.         BERKELEY
WEBSTER    ELDER   287-4598  1256, KINGFIDHER ST.    SUNNYVALE
STOP END OF JOB
```

Figure 5-4. Reading KSAM File in Chronological Sequence Using FORTRAN (continued)

6

USING KSAM FILES IN BASIC PROGRAMS

OVERVIEW

KSAM files are accessed from BASIC 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. (Refer to Table 6-1 for a list of the procedures and their associated functions.)

A KSAM file must already exist before it can be accessed from a BASIC program. It is usually created with the KSAMUTIL program BUILD command. (Refer to section II for a description of BUILD.) The BASIC procedures for accessing KSAM files do not provide a means to create a KSAM file.

Note The BASIC procedures to access KSAM files perform input-output activities differently from the BASIC 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.

Note Character substrings are expressions when used in the BASIC KSAM procedures. As such, no values can be returned to them. A copy is passed.

CALLING A KSAM PROCEDURE

The KSAM interface procedures are called from a BASIC program with a CALL statement of the following general form:

statement label `CALL` *procedure name* (*filenumber, status* `[,`*parameterlist*`]`)

Where

statement label is the number of the statement in the program.

procedure name identifies the KSAM access procedure to which control is transferred. (Refer to Table 6-1 for a complete list of the procedure names.)

filenumber is a numeric variable whose value identifies an open KSAM file. This parameter must be present. Its value is assigned by KSAM/3000 when the file is opened and must not be changed until the file is closed.

status is 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 is 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 being either variables, 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 (unlike a function, which may be assigned a value).

Refer to Table 6-1 for a complete list of the KSAM interface procedures that can be called from a BASIC program.

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*.

Table 6-1. KSAM Procedures for BASIC Interface

PROCEDURE NAME	PARAMETERS	FUNCTION	PAGE
BKCLOSE	*filenum*, *status*	Terminates processing of KSAM file identified by *filenum*.	6-8
BKDELETE	*filenum*, *status*	Logically removes record from KSAM file; the record to be deleted is the record at which the logical record pointer is currently positioned.	6-10
BKERROR	*status*, *message*	Converts numeric value returned in *status* parameter to character string *message*.	6-12
BKLOCK	*filenum*, *status* [,*condtion*]	Dynamically locks KSAM file during shared access, conditionally depending on *condition*.	6-14
BKOPEN	*filenum*, *status*, *filename* [,*access* [,*dynamic lock* [,*exclusive* [,*sequence*]]]]	Initiates processing of file identified by *filenum*, named by *filename*. Type of access, whether dynamic locking is allowed, whether access is exclusive, and whether primary key sequence is checked are options of BKOPEN.	6-16
BKREAD	*filenum*, *status* [,*parameter- list*]	Reads data from current sequential record of file identified by *filenum* into variables named in *parameterlist*.	6-22
BKREADBYKEY	*filenum*, *status*, *keyvalue*, *keylocation*, *parameterlist*	Reads data from a record identified by *keyvalue* in the key specified by *keylocation* of the file identified by *filenum* into variables named in *parameterlist*.	6-26
BKREWRITE	*filenum*, *status*, *parameterlist*	Writes data from *parameterlist* to record at which pointer is positioned in file identified by filenum.	6-29
BKSTART	*filenum*, *status* [,*keyvalue* [,*keylocation* [,*relation*]]]	Positions file identified by *filenum* in preparation for a sequential read to the first record with a key in *keylocation* whose value bears the specified *relation* to *keyvalue*.	6-32
BKUNLOCK	*filenum*, *status*	Unlocks file identified by *filenum* that has been previously locked by BKLOCK.	6-36
BKVERSION	*status*, *message*	Identifies version of KSAM/3000 currently being used returns version number in *message*.	6-38
BKWRITE	*filenum*, *status*, *parameterlist*	Writes data from *parameterlist* to record in file identified by *filenum*.	6-39

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. (Refer to Table 6-2 for possible *status* values).

Table 6-2. Values Returned to *status* Parameter

FIRST CHARACTER	REMAINING CHARACTERS
"0" successful completion	"0" no further information "2" duplicate key value
"1" at end or beginning of file	"0" no further information
"2" invalid key	"1" sequence error "2" duplicate key error "3" no record found "4" boundary violation
"7" request denied	"1" file already locked
"8" invalid call	"1" invalid number of parameters "2" invalid parameter "3" insufficient space for data in *parameterlist*
"9" file system error	"0" through "255" corresponding to file system error codes (Refer to complete list in Appendix A.)

Combining the two parts of the status code, the following values may be returned to the status parameter:

if status="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 altemate key value. Or an attempt was made by BKSTART or BKREADBYKEY to position 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 re-valued.

="*9xxx*" *File system error*—; An MPE file system error occurred for which the three-character value, *xxx* is the error code. (Refer to Table A-1 for a list of these codes.) 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$(4) <-------- dimension status string S$
      .
      . /-------------------- test first character only
      50 IF S$(1;1) = "0" THEN PRINT "SUCCESS"
      60 ELSE PRINT "ERRORCODE=";S$ <--------- print entire string
      .
      .
      . /--------------------------- test first character
     100 IF S$(1;1)= "9" THEN DO
     110 PRINT "FILE ERROR=";S$(2)
     120 DOEND \--------------------- print remaining characters
```

```
   .
   .
   .          /-------------------- test entire string
200 IFS$ = "22" THEN DO
210 PRINT "DUPLICATE KEY ERROR"
220 DOEND
300 IF S$(2)= "2" THEN PRINT "DUPLICATE KEY"
               \----------------------- test only remaining characters
```

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 key file that identifies the current record in the data file. 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. (Refer to Table 6-3 for a summary of those procedures that either position the pointer or are dependent on that position.)

Table 6-3. 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* 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.

Note BASIC procedures do not access a KSAM file in chronological sequence or by record number; they ignore the chronological 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 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 re-opens 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 successfull 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. (Refer to the Status Parameter discussion earlier in this section.) (*Required parameter*)

USING BKCLOSE

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 re-opened. Note that a KSAM file can be closed and then re-opened 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 the KSAMUTIL program.

The example in Figure 6-1, closes a file identified by the filenumber 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$)
3690 REM
3700 REM NOW DETERMINE WHETHER THTS CALL SUCCEEDED
3710 REM
3720 IF S$[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 6-1. 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. A logically deleted record is marked by two delete characters (ASCII code 255) in the first two character positions in the record. The deletion characters indicate that the record is inaccessible, although it is not physically removed from the file. The connection between a data record marked for deletion and the key file 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. (Refer to Status Parameter discussion earlier in this section.)

USING BKDELETE

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 record into variables named in the read call. When BKDELETE is successfully executed, the first two characters of the record just read are marked for deletion. Then the record is written back to the file. Any connections between the record and key entries in the key file are severed. The associated key entries are physically deleted from the key file although the data record remains in the data file. 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.

In the event that you deleted a record in error, you can recover the information in the data record by copying the data file with the NOKSAM option of FCOPY. You can copy the data file to another non-KSAM file or to the list device. With this FCOPY option, the deleted records as well as active records are copied. In order to make use of this recovery procedure, you may want to leave the first two characters of any KSAM record empty of data. In particular, you should not specify keys in those two characters.

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 file during the copy. The new file is more compact, particularly if many records had been deleted from the old file. (Refer to FCOPY description in section II for more information.)

Shared Access

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 insures that no other user alters the record position between the call that locates the record and the call that deletes it. (Refer to Table 6-3 for a list of the procedures that position the pointer and those that depend on the pointer.)

Figure 6-2 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$,A5[*],A3[*],A2[*])
3350 REM
3360 REM NOW DETERMINE WHETHER THF CALL WAS SUCCESSFUL
3370 REM
3380 IF S$[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 6-2. 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 is 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 is 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*)

USING BKERROR

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                  /---------------- full message
                /-----------------------------------\
ERROR 2 DETAIL 2 INVALID KEY VALUE. DUPLICATED KEY VALUE
ERROR 2 DETAIL 2 INVALID KEY VALUE. DUPL
                \----------------------/
                      \---------------- truncated message
```

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", then the routine above prints the following message when the program is run:

```
FILE ERROR 172 MEANS KEY NOT FOUND; NO SUCH KEY VALUE
```

A list of the MPE file system error codes and their meaning is contained in Table A-1 of appendix A.

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. (Refer to the Status Parameter discussion earlier in this section.) (*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.*

USING BKLOCK

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.

Note All users who share access to the Ele must agree to allow dynamic locking in order for any user to dynamically lock the file with BKLOCK.

The note above points out that 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 if 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 6-3 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 INDTCATES 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 6-3. Dynamically Locking a KSAM File with BKLOCK

BKOPEN

A call to procedure BKOPEN initiates 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. You can create a KSAM file through the BUILD command of the KSAMUTIL program (refer to section II).

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 and 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. (Refer to Status Parameter discussion earlier in this section.) (*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.*	*Deletes previously written data.* Use of the procedures BKREAD, BKREADBYKEY, BKREWRITE, BKDELETE, and BKSTART are prohibited.
2	*Write only.*	*Saves previously written data.* Use of the procedures prohibited by the *access*=1, above, are also prohibited by *access*=2.
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 = 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 = 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 insure 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*

USING BKOPEN

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.

Access Modes

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 insure 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 6-4 summarizes the procedures you may call depending on the *access* parameter value you specify in BKOPEN.

Table 6-4. Procedures Allowed by BKOPEN *access* Parameter

	Read-only (*access*=0)	**Write-only with Clear (*access*=1)**	**Write-only with Save (*access*=2)**	**Read/Write (*access*=3)**	**Update (*access*=4)**
Procedures Allowed	BKREAD BKREADBYKEY BKSTART BKCLOSE BKERROR	BKWRITE BKCLOSE BKERROR	BKWRITE BKCLOSE BKERROR	BKREAD BKREADBYKEY BKSTART BKWRITE BKCLOSE BKERROR	BKREAD BKREADBYKEY BKSTART BKWRITE BKREWRITE BKDELETE BKCLOSE BKERROR

Shared Access

By default in a multi-user envornment, 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 6-5 for a summary of the relation between the *exclusive* parameter and the *access* parameter.)

Table 6-5. Relation 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

Dynamic Locking

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 6-3 for a list of the pointer-dependent procedures.)

Sequence Checking

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. (Refer to the description of Status earlier in this section.) As a result of sequence checking, the chronological 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 chronological 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 chronological order.

The example in Figure 6-4 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 2 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 6-4. 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. (Refer to the Status Parameter discussion earlier in this section.) (*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.*

USING BKREAD

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 only plan to 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 = 3. If you want to rewrite or update as well as read records, you must set *access* = 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)
                / |   \-----/
              /   |       |
```

```
filenum  |      parameterlist
         |
      status
```

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") <------------ assigns numeric equivalent of string "LE" to L
```

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$) <-----------------------------------\
                                  --skip two records
220 CALL BKREAD(F,S$) <-----------------------------------/
```

The records shipped 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 (Refer to BKSTART following BKREAD discussion.)

The example in Figure 6-5 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 6-4, the records read were written in the example in Figure 6-11.

Each record contains five integers followed by five undefined words (garbage) 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$, the next two into B2$.

Shared Access

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 6-3 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 ******************************************************
 1340 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 6-5. 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. (Refer to the discussion of BKSTART.)

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. (Refer to the Status Parameter discussion earlier in this section.) (*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*)

USING BKREADBYKEY

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 there cord 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 6-6 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 6-11, 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$,A5[*],A3[*],A2[*])
2370 REM
2380 PEM NOW DETERMINE WHETHER THIS CALL HAS SUCCEEDED
2390 REM
2400 IF S$[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 6-6. 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 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. (Refer to the Status Parameter discussion earlier in this section. (*Required parameter*)

parameterlist — A list of variables or constants, separated by commas, that contain 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 so long that it exceeds the defined record length. (*Required parameter*)

USING BKREWRITE

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.

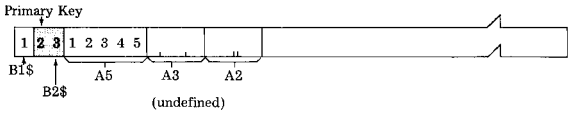
If you want to replace a record with a particular key value, you should locate and read the record with BKREADBYKEY or BKSTART. To rewrite a series of records you should read the records with BKREAD.

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 word boundaries does not occur.

The example in Figure 6-7 writes new values to a record originally written if Figure 6-11 and read in Figure 6-5. The new values fill an array that had undefined values in the last five words, 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:

Shared Access

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 insures that no other user alters the record pointer between the call that locates the record and the call that rewrites it.

Duplicate Keys

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 WRITEM 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
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
```

Figure 6-7. Rewriting Record in KSAM File with BKREWRITE

```
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
```

Figure 6-7. 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 altemate 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 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 can succeeds, to another value when it fails. (Refer to the Status Parameter discussion earlier in this section.) (*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 hey 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 1. 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*)
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.

USING BKSTART

After calling BKSTART, you should check the status parameter to determine if the procedure was executed successfully. If successfully executed, 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 exact same key value as that specified in the BKSTART call. When found, the pointer is positioned to that record. 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. For example, in a key containing a date in the form *yymmdd*, by specifying only the first two characters as *keyvalue* and a *relation* = 2, you can position to the first record with a key for that year; by specifying the first four characters, you can position to the first record for a particular year and month.

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$=" " <------------------ assign null string to keyvalue
160 L=21 <-------------------------- alternate key location to keylocation
170 CALL BKSTART(F,S$,A$,21)
```

The default for *relation* is 2 (greater than or equal to) and need not be specified except for documentation purposes.

Figure 6-8 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 6-8. Positioning Pointer to Least-Valued Record with BKSTART

The example in Figure 6-9 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 6-9. Positioning Pointer to Particular Record with BKSTART

BKUNLOCK

Unlocks a 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. (Refer to Status Parameter discussion earlier in this section.) (*Required parameter*)

USING BKUNLOCK

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 6-10 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$)
1780 REM
1790 REM NOW DETERHINE WHETHER THE CALL HAS SUCCEEDED
1800 REM
1810 IF S$(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 6-10. Dynamically Unlocking a KSAM File

BKVERSION

Retrieves the version, update number and fix number of the current KSAM/3000.

```
CALL BKVERSION(status, message).
```

A call to BKVERSION retrieves a printable string of characters that identifies the current version of the KSAM/3000 procedures used to process KSAM files. The string of characters returned by BKVERSION can be printed.

PARAMETERS

status A four-character string variable to which is returned the code that indicates whether or not the call to BKVERSION was successful and if not, why not. The first character is set to zero when the call succeeds, to another value if it fails. (Refer to the Status Parameter discussion earlier in this section.) (*Required parameter*)

message A string variable to which is returned the identification of the current KSAM/3000 procedures. It is in the form:

- *version.update.fix*

where *version* is an ASCIlletter, *update* is an ASCII integer, and *fix* is also an ASCII integer. The three terms are separated by periods. (*Required parameter*)

USING BKVERSION

You may call BKVERSION in order to get the version, update, and fix numbers of the KSAM/3000 currently being used. This identification can be compared to the version, update, and fix numbers that identify the version in which a KSAM file was created, as returned by the VERIFY command of program KSAMUTIL (refer to section II). The following example illustrates use of BKVERSION. Note that two strings are needed.

```
10 DIM S$(4) <-------- status
20 DIM V$(72) <---- message
:
100 CALL BKVERSION(S$,V$)
110 PRINT "THE CURRENT KSAM/3000 IS HP32208.";V$
:
RUN
THE CURRENT KSAM/3000 IS HP32208.A.1.23
                                  \-----/
                                     |
                               contents of V$
```

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. (Refer to the Status Parameter discussion earlier in this section.) (*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*)

USING BKWRITE

After calling BKWRITE, you should always check the *status* parameter to insure 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.

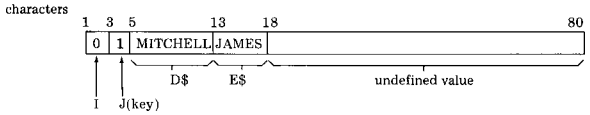
The *sequence* parameter determines whether records must be written in primary key sequence, or not. If *sequence* is zero, records can be written in any order; no check is made on the sequence of the primary key field. If *sequence* is set to 1, you must write each record with a value in the primary 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. To illustrate, assume that the record illustrated by the following example was the first record written to the file. It has the value 1 as its primary key. If the file was opened with *sequence* = 1, the next record written must have a

value of 2 or more in the primary key field. It may have the same value only if duplicates are allowed for that key field, and must not have a value less than the previous primary key.

The values written to the record depend on the type of the items in *parameterlist*. To illustrate, consider the following statements:

```
10 DIM D$(20),E$(10),S$(4)
20 INTEGER I,J
30 D$="MITCHELL"<------------------ logical length = 8 characters
40 E$="JAMES" <------------------------ logical length = 5 characters
50 I=0<------------|
    each integer requires 2 characters
60 J=1<------------|
70 CALL BKWRITE (F,S$,I,J,D$,E$)
                 /  |  \-------/
                /   |     |
       filnum       |     parameterlist
                 status
```

This set of statements writes one record to the KSAM file. The record has the form:



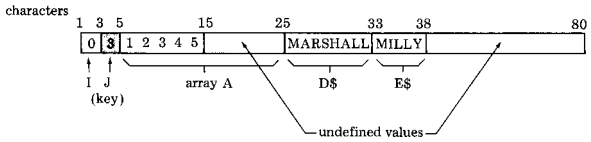
Assuming a file created with one key starting in the third character, two characters long, the value 1 is the key value. Each integer requires 2 characters, the two strings use a total of 13 characters, resulting in values that take up 17 characters of the record. The remainder of the record is undefined. Record size is specified at file creation.

When writing from numeric arrays, the dimensioned length is used; when writing from strings the logical length is used. The logical length of a string variable or string array element, is the number of characters actually stored in the variable or element. It determines the length of the item written to the record. A numeric array, on the other hand, uses the dimensioned length as the length of the item written to the record. For example, suppose a numeric array A is added to the *parameterlist* in the previous example:

```
 5 INTEGER A(10) <------------------------ dimensioned length of A is 10 words
10 DIM D$(20),E$(10),S$(4)
20 INTEGER I,J,F
30 D$="MARSHALL"
40 D$="MILLY"
50 FOR I=1 TO 5 \
60 A(I)=1 |--------------------- Move 5 words to array A
70 NEXT I &/
80 I=0
```

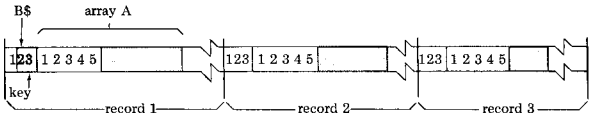
```
 90 J=3
100 CALL BKWRITE(F,S$,I,J,A(*),D$,E$)
```

This set of statements results in a record with the following values:



Note Items written to a KSAM file from a BASIC program are concatenated; rounding to word boundaries does not occur.

Figure 6-11 is an example of writing one string and one integer array to each record of the KSAM file opened in Figure 6-4. The three records written contain the following data:



```
 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 PEM KEY=B,2,2,DUP
120 REM SO,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 PEM 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

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
```

Figure 6-11. Writing to a KSAM File with BKWRITE

```
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
```

Figure 6-11. Writing to a KSAM File with BKWRITE (continued)

A

ERROR MESSAGES AND RECOVERY PROCEDURES

This appendix lists the error messages that may be issued as a result of errors encountered while accessing KSAM files. The messages are not limited to KSAM errors since other file system errors or language errors can occur while accessing a KSAM file.

Whenever possible, the reason the message was issued is listed under "Meaning" and any action that can be taken to correct the error is listed under "Action."

The messages are contained in the following tables:

Table A-1. File System Error Codes

CODE	MEANING	ACTION
0	END OF FILE	
1	ILLEGAL DB REGISTER	
2	ILLEGAL CAPABILITY	
3	OMITTED PARAMETER	
4	INCORRECT S REGISTER	
5	PARAMETER ADDRESS VIOLATION	
6	PARAMETER END ADDRESS VIOLATION	
7	ILLEGAL PARAMETER	
8	PARAMETER VALUE INVALID	
9	INCORRECT Q REGISTER	
20	INVALID OPERATION	
21	DATA PARITY ERROR	
22	SOFTWARE TIME-OUT	
23	END OF TAPE	
24	UNIT NOT READY	
25	NO WRITE-RING ON TAPE	
26	TRANSMISSION ERROR	
27	I/O TIME-OUT	
28	TIMING ERROR OR DATA OVERRUN	
29	SIO FAILURE	
30	UNIT FAILURE	
31	END OF LINE	
32	SOFTWARE ABORT	
33	DATA LOST	
34	UNIT NOT ON-LINE	
35	DATA-SET NOT READY	
36	INVALID DISC ADDRESS	
37	INVALID MEMORY ADDRESS	
38	TAPE PARITY ERROR	
39	RECOVERED TAPE ERROR	
40	OPERATION INCONSISTENT WITH ACCESS TYPE	
41	OPERATION INCONSISTENT WITH RECORD TYPE	
42	OPERATION INCONSISTENT WITH DEVICE TYPE	
43	WRITE EXCEEDS RECORD SIZE	
44	UPDATE AT RECORD ZERO	
45	PRIVILEGED FILE VIOLATION	
46	OUT OF DISC SPACE	
47	I/O ERROR ON FILE LABEL	
48	INVALID OPERATION DUE TO MULTIPLE FILE ACCESS	
49	UNIMPLEMENTED FUNCTION	
50	NONEXISTENT ACCOUNT	

Table A-1. File System Error Codes (continued)

CODE	MEANING	ACTION
51	NONEXISTENT GROUP	
52	NONEXISTENT PERMANENT FILE	
53	NONEXISTENT TEMPORARY FILE	
54	INVALID FILE REFERENCE	
55	DEVICE UNAVAILABLE	
56	INVALID DEVICE SPECIFICATION	
57	OUT OF VIRTUAL MEMORY	
58	NO PASSED FILE	
59	STANDARD LABEL VIOLATION	
60	GLOBAL RIN UNAVAILABLE	
61	OUT OF GROUP DISC SPACE	
62	OUT OF ACCOUNT DISC SPACE	
63	USER LACKS NON-SHARABLE DEVICE CAPABILITY	
64	USER LACKS MULTI-RIN CAPABILITY	
71	TOO MANY FILES OPEN	
72	INVALID FILE NUMBER	
73	BOUNDS VIOLATION	
80	SPOOFLE SIZE EXCEEDS CONFIGURATION	
81	NO "SPOOL" CLASS IN SYSTEM	
82	INSUFFICIENT SPACE FOR SPOOFLE	
83	I/O ERROR ON SPOOFLE	
84	DEVICE UNAVAILABLE FOR SPOOFLE	
85	OPERATION INCONSISTENT WITH SPOOLING	
86	NONEXISTENT SPOOFLE	
87	BAD SPOOFLE BLOCK	
89	POWER FAILURE	
90	EXCLUSIVE VIOLATION: FILE BEING ACCESSED	
91	EXCLUSIVE VIOLATION: FILE ACCESSED EXCLUSIVELY	
92	LOCKWORD VIOLATION	
93	SECURITY VIOLATION	
94	USER IS NOT CREATOR	
100	DUPLICATE PERMANENT FILE NAME	
101	DUPLICATE TEMPORARY FILE NAME	
102	I/O ERROR ON DIRECTORY	
103	PERMANENT FILE DIRECTORY OVERFLOW	
104	TEMPORARY FILE DIRECTORY OVERFLOW	
106	EXTENT SIZE EXCEEDS MAXIMUM	
107	INSUFFICIENT SPACE FOR USER LABELS	
108	DEFECTIVE FILE LABEL ON DISC	
110	ATTEMPT TO SAVE PERMANENT FILE AS TEMPORARY	

Table A-1. File System Error Codes (continued)

CODE	MEANING	ACTION
111	USER LACKS SAVE FILE CAPABILITY	
112 ⋮ 169	RESERVED FOR FUTURE USE	
170	THE RECORD IS MARKED DELETED. FPOINT POSITIONED POINTER TO A RECORD THAT WAS MARKED FOR DELETION.	
171	DUPLICATE KEY VALUE WHEN DUPLICATES NOT ALLOWED.	
172	KEY NOT FOUND; NO SUCH KEY VALUE.	
173	*tcount* PARAMETER LARGER THAN RECORD SIZE.	
174	CANNOT GET EXTRA DATA SEGMENT FOR THIS FILE.	
175	KSAM INTERNAL ERROR. A KEY VALUE (NOT SEARCH KEY) FOR A RECORD TO BE DELETED IS NOT IN KEY FILE; RECORD CANNOT BE DELETED.	
176	ILLEGAL EXTRA DATA SEGMENT LENGTH.	
177	TOO MANY EXTRA DATA SEGMENTS FOR THIS PROCESS	
178	NOT ENOUGH VIRTUAL MEMORY FOR EXTRA DATA SEGMENT	INCREASE THE SIZE OF VIRTUAL MEMORY
179	THE FILE MUST BE LOCKED BEFORE THIS INTRINSIC ISSUED	USE FLOCK TO LOCK FILE OR OPEN FILE FOR EXCLUSIVE ACCESS.
180	THE KSAM FILE MUST BE REBUILT BECAUSE THIS VERSION OF KSAM DOES NOT HANDLE THE FILE BUILT BY PREVIOUS VERSION.	USE FCOPY TO REBUILD FILE: >FROM=*oldksamfile* ;TO=(*dfile*,*kfile*)
181	INVALID KEY STARTING POSITION.	
182	FILE IS EMPTY.	
183	RECORD DOES NOT CONTAIN ALL THE KEYS.	
184	INVALID RECORD NUMBER. RECORD NUMBER IS NEGATIVE, OUT OF RANGE, OR DELETED.	RECORD NUMBER MUST BE POSITIVE INTEGER.
185	SEQUENCE ERROR IN PRIMARY KEY; ATTEMPT TO WRITE RECORD WITH PRIMARY KEY LESS THAN PREVIOUS KEY WHEN ASCENDING SEQUENCE EXPECTED.	
186	INVALID KEY LENGTH.	
187	INVALID KEY SPECIFICATION; KEYS ILLEGAL.	
188	INVALID DEVICE SPECIFICATION.	
189	INVALID RECORD FORMAT.	

Table A-1. File System Error Codes (continued)

CODE	MEANING	ACTION
190	INVALID KEY BLOCKING FACTOR VALUE.	
191	RECORD DOES NOT CONTAIN SEARCH KEY FOR DELETION. SPECIFIED KEY VALUE POINTS TO RECORD WHICH DOES NOT CONTAIN THAT VALUE.	
192	SYSTEM FAILURE OCCURRED WHILE KSAM FILE WAS OPENED	RUN KEY INFO OF KSAMUTIL TO RESET FLAG.
193	$STDIN/$STDINX CANNOT BE REDIRECTED TO KSAM FILES.	
194	KSAM FILES NOT ALLOWED FOR GLOBAL AFT'S.	
195	GLOBAL FILES CANNOT BE REMOTE FILES.	
196	LANGUAGE NOT SUPPORTED.	
197	NATIVE LANGUAGE INTERNAL ERROR.	
198	INVALID VERSION NUMBER IN KSAM FILE.	
193 ⋮ 200	RESERVED FOR KSAM	
201 ⋮ 255	RESERVED FOR FUTURE USE	

Table A-2. COBOL Status Parameter Return Values

STATUS VALUE	MEANING	ACTION
"00"	SUCCESSFUL COMPLETION—I/O operation was completed successfully.	None.
"02"	SUCCESSFUL COMPLETION, DUPLICATE KEY—Read or Readbykey read a record whose key value was the same as the equivalent key in the next sequential record; this is not an error since duplicate alternate keys are allowed. Write or rewrite operation was successful; a duplicate key was written for a key that is allowed duplicates.	None required, returned for information only.
"10"	AT END—End-of-file or beginning-of-file reached during sequential or random read. There is no next logical record in ascending key order.	Usually none. This result is a signal to close the file or perform another end-of-file action.
"21"	INVALID KEY, SEQUENCE ERROR—Attempt was made to write a record with a primary key that is out of sequence when the file was opened for sequential access.	Check the primary key value in the record being written. If you don't want sequence checking, re-open the file for random or dynamic access.
"22"	INVALID KEY, DUPLICATE KEY—Attempt was made to write or rewrite a record with a key value that duplicates a key value in an existing record, and duplicates are not allowed.	Check the key values. If possible change them to avoid the duplication. If duplicate keys must be written, create the file again allowing duplicates for the key and then copy the old file to the new file with FCOPY.
"23"	INVALID KEY, NO RECORD FOUND—Attempt to access record identified by a key with CKSTART or CKR EADBY KEY, but no record is found with the specified key value at the specified key location.	Check the *keyvalue*, *keylength*, an *keylocation* parameters in the call, Correct if necessary. If record that cannot be found should be in the file, you may want to list the data file with FCOPY.
"24"	INVALID KEY, BOUNDARY VIOLATION—An attempt was made to write beyond the externally defined end of file.	
"30"	LOCK DENIED—File was locked by another process.	Wait until process locking file unlocks it—try again or lock file with *lockcond* = 1.

Table A-2. COBOL Status Parameter Return Values (continued)

STATUS VALUE	MEANING	ACTION
"31"	UNLOCK DENIED—File was not locked by calling process.	Before calling CKUN LOCK to unlock a shared file it must have been locked by a call to CK LOCK.
"9n"	FILE SYSTEM ERROR—Where *n* is a binary number between 0 and 255 corresponding to a File System Error code (Refer to Table A-1).	Within your program you can call CKERROR to convert the number to a displayable value and then display it. Look up the value in Table A-1 and perform any suggested action.

Note that COBOL error messages 752 and 753 are issued for errors processing KSAM files. (Refer to Table C-2 in the COBOL manual.)

Table A-3. BASIC Status Parameter Return Values

STATUS	MEANING	ACTION
"00"	SUCCESSFUL COMPLETION—The current I/O operation was completed successfully.	None.
"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. Or 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.	None required. Returned for information only.
"10"	AT END—A sequential read was attempted with BKREAD, but there was no next logical record in ascending sequence by key value, or random read attempted to position to record with key value less than lowest value or greater than greatest value.	Usually none. This result is a signal to close the file or perform some other end-of-file function.
"21"	INVALID KEY, SEOUENCE ERROR—BKWRITE attempted; record being written has primary key that is not in sequential order but file was opened for sequence checking. BK REWRITE attempted, but the primary key value was changed since the record being rewritten was read.	Check the primary key value in the record; if you don't want sequence checking, reopen the file with *sequence* = 0. Check the primary key value. Either change it back to the original value or read the record again before calling BKREWRITE.
"22"	INVALID KEY, DUPLICATE KEY ERROR—BKWRITE or BKREWRITE attempted to write record that contains a duplicate value for a key that is not allowed duplicate values.	Check the key value. If possible change it to a unique value. If duplicate keys must be written, create the file again allowing duplicate values for the key and then copy the old file to the new file.
"23"	INVALID KEY, NO RECORD FOUND—BKSTART or BKR EADBYKEY attempted to locate a record by a key value that could not be found.	Check the key value, and key location parameters. Correct if necessary. If record that cannot be found should be in file, you may want to list data file with FCOPY.
"24"	INVALID KEY, BOUNDARY VIOLATION—BKWRITE attempted to write beyond the externally defined boundaries of the file.	Re-enter command.

Table A-3. BASIC Status Parameter Return Values (continued)

STATUS	MEANING	ACTION
"71"	REQUEST DENIED, FILE ALREADY LOCKED—BK LOCK was called with conditional locking and the file was already locked by another user.	Perform some action that does not require exclusive access and then try BK LOCK again. As soon as the other user unlocks the file BKLOCK will work.
"81"	INVALID CALL, WRONG NUMBER OF PARAMETERS—A procedure call had too many or too few parameters.	Check the call syntax and correct; if not sure of error, consult manual.
"82"	INVALID CALL, INVALID PARAMETER—Specified parameter is not the correct type. For example, a string variable was specified where only numeric variables or constants are allowed.	Check the parameter and compare it to the rules specified in this manual for the parameter format.
"83"	INVALID CALL, INSUFFICIENT INTERNAL BUFFER SPACE—The data to be written to the file is too long for the configured internal buffer space.	You can ask the system manager to reconfigure the system, or you may be able to reduce the amount of data being written from *parameterlist*
"9*xxx*"	FILE SYSTEM ERROR—An MPE file system error occurred for which the value *xxx* specifies a 3-digit code between "0" and "255".	Within your program you can call BKER ROR to display a message with the meaning of the code; you can consult Table A-1 for the code meaning.

Table A-4. KSAMUTIL Error Codes and Messages

CODE	MESSAGE	MEANING	ACTION
1000	COMMAND FILE READ ERROR.		
1001	COMMAND FILE END OF FILE.		
1002	UNKNOWN COMMAND. TYPE HELP.	Command name not recognized.	Type the correct name if you know it; else type HELP.
1003	TOO MANY PARAMETERS FOR THIS COMMAND.	More parameters specified than are allowed.	Check the command syntax and reenter correctly.
1004	COMMAND FILE DATA TRANSMISSION ERROR.		
1005	COMMAND TOO LONG.		
1006	FILE NAME TOO LONG OR ABSENT.	File name in BUILD command incorrect.	Enter file name of 8 or fewer alphanumeric characters starting with a letter.
1007	'REC' PARAMETER LIST EXHAUSTED.	Too many parameters specified after REC= in BUILD command.	Check syntax and reenter with correct number of parameters.
1008	'REC' RECORD SIZE VALUE INVALID.	Record size in BUILD command is not valid.	Check syntax and reenter with correct value for record size.
1009	'REC' BLOCKING FACTOR VALUE INVALID.	Block factor in BUILD command is not valid.	Check syntax and reenter with correct value for blocking factor.
1010	'REC' RECORD FORMAT VALUE INVALID.	Record format in BUILD command is not F or V.	Enter V for variable length records, F for fixed length, or omit for fixed length.
1011	'REC' RECORD TYPE VALUE INVALID.	Record code in BUILD command is not ASCII or BINARY.	Enter ASCII for ASCIIcode records; BINARY or omit for binary-coded records.
1012	'DEV' DEVICE VALUE ABSENT.	No device specification after DEV= in BUILD command.	Enter legal device name or omit if device class is DISC.

Table A-4. KSAMUTIL Error Codes and Messages (continued)

CODE	MESSAGE	MEANING	ACTION
1013	'DEV' DEVICE VALUE INVALID.	Device specification in BUILD command is not valid.	Enter legal device class name or legal logical device number. Refer to *System Manager/System Supervisor Manual.*
1014	'DEV' DEVICE VALUE TOO LONG OR ABSENT.	DEV= missing device specification or one that is too long.	Enter device name that is 1 to 8 alphanumeric characters, begins with a letter, terminates with nonalphanumeric character; or omit DEV= for DISC; or enter logical device number.
1015	'KEY' SPECIFICATIONS EXCEED LIMIT.	Too many KEY= specifications in BUILD.	Reduce number of keys to a total of 16.
1016	'KEY' CONTAINS TOO MANY PARAMETERS.	More than 5 parameters follow KEY= in BUILD.	Parameters are: *keytype*, *keylocation*, *keysize*, and optionally, *keyblocking*, and DUP or DUPLICATE.
1017	'KEY' TYPE VALUE INVALID.	Invalid key type specified after KEY= in BUILD.	Key types may be: B, I, D R, L, N, P,*. Enter one of correct types.
1018	'KEY' POSITION VALUE ABSENT.	The key location is missing from KEY= in BUILD command.	Enter required *keylocation* parameter following *keytype* type in KEY= specification.
1019	'KEY' POSITION VALUE INVALID.	Invalid keylocation parameter specified after KEY= in BUILD.	Key location is specified as integer between 1 and number of bytes in record.
1020	'KEY' SIZE VALUE REQUIRED AND ABSENT.	The key size is missing from KEY= in BUILD command.	Enter required *keysize* parameter after *keylocation* in KEY= specification.
1021	'KEY' SIZE VALUE INVALID.	Invalid keysize parameter specified after KEY= in BUILD.	Key size is specified as the number of bytes in the key; refer to table 02-02 for legal sizes for each key type.

Table A-4. KSAMUTIL Error Codes and Messages (continued)

CODE	MESSAGE	MEANING	ACTION
1022	'KEY' BLOCKING FACTOR VALUE INVALID.	Invalid key blocking parameter specified after KEY= in BUILD.	Specify *keyblocking* as an even number equal to or greater than 4; or omit for key blocks with four keys per block.
1023	CONFLICTING OPTIONS		Check command syntax
1024	MISSING CLOSING QUOTE		Check command syntax
1025	'KEY' 'DUPLICATE' EXPECTED.	Key word DUP or DUPLICATE expected in KEY= specification.	Enter DUP or DUPLICATE or remove terminating commas.
1026	'DISC' MUST BE FOLLOWED BY '='.	Key word DISC in BUILD command was specified without =.	Reenter DISC= followed by up to 3 parameters describing disc file, or omit for defaults.
1027	'DISC' NUMBER OF RECORDS VALUE INVALID.	Value of *numrecs* parameter to DISC= not a positive integer.	Enter maximum number of records as file size, or omit for default value of 1023 records.
1028	'DISC' NUMBER OF EXTENTS VALUE INVALID.	Value of *numextents* not in range 1-32.	Enter integer between 1 and 32, or omit for default value of 8 extents.
1029	'DISC' INITIAL ALLOCATION VALUE INVALID.	Value of *initalloc* not in range 1-32.	Enter integer between 1 and 32, or omit for default value of 1 extent allocated when file is opened.
1030	'LABELS' NOT FOLLOWED BY '=' OR BY TOO MANY PARAMETERS.	Key word LABELS in BUILD command was specified without = or had more than 1 parameter.	Reenter LABELS= followed by one parameter to specify number of user labels, or omit for default of 0.
1031	'LABELS' NUMBER OF LABELS VALUE INVALID.		

Table A-4. KSAMUTIL Error Codes and Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
1032	'FIRSTREC' NOT FOLLOWED BY '=' OR BY TOO MANY PARAMETERS.	Key word FIRSTREC in BUILD command was specified without = or had more than 1 parameter.	Reenter FIRSTREC=followed by starting record number, or omit to start numbering records with zero.
1033	'FIRSTREC'STARTING RECORD NUMBER MUST BE 0 OR 1.	Value other than 0 or 1 entered for first record number.	Enter correct value or omit for default of 0.
1034	'CODE' NOT FOLLOWED BY '=' OR BY TOO MANY PARAMETERS.	Key word CODE in BUILD command was specified without = or had more than one parameter.	Reenter CODE = followed by *filecode*, or omit for file code of zero.
1035	'CODE' FILE NUMBER VALUE INVALID.	Value not in range 0 through 1023 entered for file code.	Enter positive integer between 0 and 1023, or omit for default of 0.
1036	'KEYDEV' FOLLOWED BY TOO MANY PARAMETERS.	Key word KEYDEV= in BUILD command was specified with more than 1 parameter.	Reenter with one parameter to specify device class or logical device number of key file, or omit for DISC.
1037	'KEYDEV' MUST BE FOLLOWED BY'='.	Key word KEYDEV= not followed by =.	Reenter KEYDEV=, or omit for default device class DISC.
1038	'KEYDEV' DEVICE PARAMETER VALUE TOO LONG OR ABSENT.	Key word KEYDEV= must be followed by valid device parameter.	Reenter with device class specified as 1 to 8 alphanumeric characters beginning with letter, terminated by non-alphanumeric character, or reenter with logical device number, or omit for default DISC.
1039	'KEYFILE' NOT FOLLOWED BY '=' OR BY TOO MANY PARAMETERS.	Key word KEYFILE in BUILD command was specified without = or had more than 1 parameter.	Reenter KEYENTITY = followed by actual file designator of key file.
1040	KEY FILE NAME TOO LONG OR ABSENT.	File name specified as KEYENTITY = parameter is more than 8 characters or was omitted.	Reenter KEYENTITY = with correct file name format. (Refer to BUILD description in manual.)

Table A-4. KSAMUTIL Error Codes and Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
1041	'KEYENTRIES' NOT FOLLOWED BY '=' OR BY TOO MANY PARAMETERS.	Keyword KEYENTR IES in BUILD command was specified without =, or had more than 1 parameter.	Reenter KEYENTRIES= followed by the maximum number of primary key entries expected in the key file, or omit for default of *numrecs* value from REC=parameter.
1042	'KEYENTRIES' NUMBER OF		ENTRIES VALUE INVALID.
1043	KEYWORD SPECIFICATION IN THIS COMMAND IS INVALID.	A key word specified as a KSAMUTIL command parameter is misspelled or not in syntax.	Check command syntax for correct key word and/or spelling; reenter correctly.
1044	DELIMITER AT THE END OF A SPECIFICATION IS INVALID.	A delimiter follows command specification in KSAMUTIL command.	Remove delimiter or follow with rest of command.
1045	THE NUMBER OF PARAMETERS IN THIS COMMAND IS INVALID.	Too many or too few parameters specified in a KSAMUTIL command.	Check command syntax and reenter with the correct number of parameters.

Table A-4. KSAMUTIL Error Codes and Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
1046	A PARAMETER VALUE IS INVALID. 'TEMP'WAS EXPECTED.	The key word TEMP was expected as a parameter in PURGE or RENAME commands.	Check command syntax, reenter with correct parameter value.
1047	LOCKWORD NOT ALLOWED IN KEY FILE		Check command syntax; remove lockword from key file specification.
1048	GROUP AND/OR ACCOUNT NOT ALLOWED FOR KEYENTITY =		Check command syntax of BUILD command.
1049	BACK REFERENCE NOT ALLOWED ON *formal-designator*		
1050	SEQ=SEQUENCE NUMBER IS INVALID	Non-numeric sequence number specified for SEQ= parameter.	Check command syntax of KEYSEQ or KEYDUMP commands.
1051	SUBSET= VALUE INVALID	Either the starting position or the number of key values to be dumped is invalid.	Check command syntax of KEYDUMP command
1052	SEQ= SYNTAX ERROR		Check command syntax of KEYSEQ or KEYINFO
1053	SEQ= PARAMETER LIST EXHAUSTED	Not enough information in parameter list; key number missing.	Enter parameter value for SEQ= in KEYSEQ or KEYDUMP commands.
1054	ENTITY = PARAMETER LIST EXHAUSTED	Not enough information in parameter list; file name missing.	Enter file name after ENTITY = in KEYDUMP command.
1055	ENTITY = SYNTAX ERROR		Check command syntax of KEYDUMP command.
1056	SUBSET= PARAMETER LIST EXHAUSTED	Not enough information in parameter list.	Check command syntax of KEYDUMP; enter correct number of parameters.

Table A-4. KSAMUTIL Error Codes and Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
1057	SUBSET= SYNTAX ERROR		Check command syntax of KEYDUMP command.
1058	INVALID KEY SEQUENCE SPECIFICATION	Key number specified in SEQ= parameter is greater than the number of keys in file.	Key number is 1 for primary key, 2 for first alternate key, etc. Use VERIFY command to check number of keys in file.
1059	FILE SPECIFIED IN FILE-ALREADY EXISTS	File name specified in ENTITY = parameter is an existing permanent file; KEYDUMP always creates a new file.	Enter name of non-existent file, or rename permanent file.
1060	B-TREE HAS MORE THAN 20 LEVELS	KEYDUMP cannot dump more than 20 levels of the key file structure.	
1061	INVALID DECIMAL DIGIT OR DIGIT COUNT >28	Packed decimal digit is not 0-9, or there are more than 28 digits. Cannot convert to ASCII for KEYDUMP.	
1062	THE REFERENCED FILE IS NOT A KSAM FILE	File reference in the command is not a KSAM file.	Check file name and correct it.
1063	RECORD SIZE OF THE SPECIFIED FILE HAS BEEN CHANGED	Record size of the file specified in ENTITY = parameter of KEYDUMP has been changed by a : FILE command.	Check KEYDUMP syntax; change : FILE command so that record size is not specified.
1064	GENERIC OR APPROXIMATE SEARCH NOT ALLOWED FOR KEY TYPE	Generic or approximate keys can be specified in SUBSET= parameter of KEYDUMP only if key type is BYTE, INTEGER, or DOUBLE.	Use full key value in SUBSET= or do not use SUBSET= parameter.

Table A-4. KSAMUTIL Error Codes and Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
1065	ILLEGAL OR TOO MANY CHARACTERS	*Position* value of SUBSETparameter in KEYDUMP contains non-numeric characters, or is >9 digits.	Change position value, or use quoted string for SUBSET= parameter.
1066	REMOTE FILE ACCESS NOT SUPPORTED	A :FILE command specified a remote file, but this command does not support remote access.	Change : FILE command to specify local file.
1067	SORT ON RECORD POINTERS FAILS	Sort of keydump by record pointers (SORT option of KEYDUMP) failed during sort by SORT/3000 program.	Check the reasons for failure in SORT error message.
1068	SYSTEM FAILURE OCCURRED WHILE THE KSAM FILE WAS OPEN	KSAM file was open when a system failure occurred, and file may be damaged.	Run KEYINFO command to recover file and reset flag so file can be opened, or run VERIFY with NOCHECK to examine file.
1069	UNEXPECTED CHARACTER IN FILE NAME; EXPECTED . or /	The only nonmeric characters allowed in a file name are "." or "/".	Check the file name and correct it.

Table A-5. FCOPY Warning and Error Messages

CODE	MESSAGE	MEANING	ACTION
None	<CONTROL Y>	Acknowledges receipt of a CONTROL-Y entered during a session.	None.
None	READ ERROR FROM COMMAND INPUTFILE	An error occurred while reading an FCOPY command from $STDIN.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
None	WRITE ERROR TO COMMAND LISTFILE	An error occurred while writing an FCOPY message to $STDLIST.	More than likely nothing serious has occurred and all FCOPY operations have been performed successfully. If you want to be sure, however, do the following: *In a job*: Re-submit the job. *In a session*: Re-enter the most recent FCOPY command.
3	SYNTAX ERROR: IN SUBSET OPTION	The subset function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
4	SYNTAX ERROR: IN TITLE OPTION	The title option of the display function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
5	SYNTAX ERROR: IN IGNERR OPTION	The ignore errors function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
6	SYNTAX ERROR: IN VERIFY OPTION	The verify function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
7	SYNTAX ERROR: IN SKIPEOF OPTION	The skip end-of-file function was not specified properly	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format
8	SYNTAX ERROR: IN COMPARE OPTION	The compare function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
9	SYNTAX ERROR: IN NEW OPTION	The new file function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
10	SYNTAX ERROR: IN HEX OPTION	The display hexadecimal function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
11	SYNTAX ERROR: IN EBCDICOUT OPTION	The EBCDICOUT character translate function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
12	SYNTAX ERROR: IN CHAR OPTION	The display character function was not specified properly	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
13	SYNTAX ERROR: IN OCTAL OPTION	The display octal function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
14	SYNTAX ERROR: IN UPSHIFT OPTION	The upshift function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
15	SYNTAX ERROR: IN BCDICIN OPTION	The BCDICIN character translate was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
16	SYNTAX ERROR: IN NORECNUM OPTION	The NORECNUM option of the display function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
17	SYNTAX ERROR: IN EBCDICIN OPTION	The EBCDICIN character translate function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
18	SYNTAX ERROR: IN BCDICOUT OPTION	The BCDICOUT character translate function was not specified properly.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
19	SYNTAX ERROR: INVALID FORM OF EXIT COMMAND	The EXIT command was not specified properly.	None. FCOPY terminates

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
51	SYNTAX ERROR: IN QUOTED STRING	The *characterstring* specified for the subset function is not valid.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
52	SYNTAX ERROR: IN BIT PATTERN	The *patternlist* specified for the subset function is not valid.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
53	SYNTAX ERROR: INVALID INTEGER	An integer specified is outside the range allowed for the particular FCOPY function.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
54	SYNTAX ERROR: UNKNOWN OPTION NAME	One of the specified functions was unrecognizable.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
55	SYNTAX ERROR: IN FROMFILE SPECIFIER	The "from" file was not specified properly.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
56	SYNTAX ERROR: IN TOFILE SPECIFIER	The "to" file was not specified properly.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
57	SYNTAX ERROR: ILLEGAL COMBINATION OF OPTIONS	Two or more *functionlist* entries conflict with one another.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
58	SYNTAX ERROR: FROMFILE AND TOFILE NOT BOTH SPECIFIED	FROM= and TO= were not both specified in the FCOPY command.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
59	SYNTAX ERROR: ILLEGAL USE OF NEW	The context used to specify a new "to" file is not valid.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
60	SYNTAX ERROR: ILLEGAL USE OF *	The context used to specify* as a "from" file or "to" file is not valid.	*In a job*: Correct the command and resubmit the job, *In a session*: Re-enter the command using the correct format.
62	SYNTAX ERROR: FILE NAME TOO LONG	The "from" or "to" file name specified is longer than the 35 characters allowed in a fullyqualified file name with lockword.	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
102	CAN'T CLOSE FROMFILE	MPE can't close the "from" file. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.
103	CAN'T CLOSE TOFILE	MPE can't close the "to" file. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
104	CAN'T SAVE NEW TOFILE	MPE can't close the "to" file as a permanent file. Either you do not have SF capability or there is not enough group account, or system file space.	If you don't have SF capability, you can't perform the operation. If there is not enough file space, purge some unneeded files to free some file space.
105	CAN'T OPEN FROMFILE	MPE can't open the "from" file. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.
106	CAN'T OPEN TOFILE	MPE can't open the "to" file. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.
107	VERIFY OR COMPARE OPTION: CAN'T GET READ ACCESS TO TO FILE	MPE can't get read access to the "to" file for a verify or compare operation. The read access specified in the file label has been overridden by an MPE :FILE command containing ACC=APPEND, ACC=OUT, or ACC=OUTKEEP.	Reset the particular : FILE command (using the MPE : RESET command) and retry the operation.
108	ERROR IN CALLING FGETINFO FOR FROMFILE	An error prevented MPE from obtaining information from the "from" file's label. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.

Table A-5. FCOPY Warning and Error Messages (continued)

CODE	MESSAGE	MEANING	ACTION
109	ERROR IN CALLING FGETINFOR FOR TOFILE	An error prevented MPE from obtaining information from the "to" file's label. This message is followed by an MPE file information display containing (among other things) an error number.	Look up the error number in Table A-1 and act accordingly.
110	IGNERR OPTION: FROM FILE NOT TAPE	The "from" file's device is not a magnetic tape unit.	The ignore errors function cannot be used in this case.
111	CAN'T GET READ ACCESS TO FROMFILE	MPE can't get read access to the "from" file. The read access specified in the file label has been overridden by an MPE :FILE command containing ACC=APPEND, ACC=OUT, or ACC=OUTKEEP.	Reset the particular : FILE command (using the MPE : RESET command) and retry the operation.
112	CAN'T GET WRITE ACCESS TO TO FILE	MPE can't get write access to the "to" file. The write access specified in the file label has been overridden by an MPE :FILE command containing ACC=IN.	Reset the particular : FILE command (using the MPE : RESET command) and retry the operation.
113	SKIPEOF OPTION: FROM FILE NOT TAPE	The skip end-of-file function was specified for the "from" file and the "from" file device is not a magnetic tape unit.	If the intended "from" or "to" file is on magnetic tape, check the associated MPE :FILE command and the back reference to it.
114	SKIPEOF OPTION: TOFILE NOT TAPE	The skip end-of-file function was specified for the "to" file and the "to" file device is not a magnetic tape unit.	If the intended "from" or "to" file is on magnetic tape, check the associated MPE :FILE command and the back reference to it.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
115	SUBSET OPTION: STRING FALLS OUTSIDE OF FROMFILE RECSIZE	The *characterstring* or *patternlist* specified is greater than the record size of the "from" file. No such subset can exist in the specified "from" file.	Change the specified subset definition to a valid one and try the operation again.
116	CAN'T GET LARGE ENOUGH BUFFER	There is not enough data space for the buffers needed by the requested operation. FCOPY uses the DL-DB area for variable sized buffers.	Ask the system manager what size data area was specified when FCOPY was prepared and rerun FCOPY specifying a larger MAXDATA= parameter. Also make sure that the system configuration will accommodate your record size in the maximum allowed data segment size.
117	SKIPEOF OPTION: ERROR WHILE SKIPPING IN FROMFILE	An error occurred while skipping end-of-file marks in the "from" file.	Retry the operation.
118	SKIPEOF OPTION: ERROR WHILE SKIPPING IN TOFILE	An error occurred while skipping end-of-file marks in the "to" file.	Retry the operation.
119	SUBSET OPTION: ERROR WHILE SPACING IN FROMFILE	An error occurred while spacing through the "from" file.	Retry the operation.
120	SUBSET OPTION: SUBSET STARTS OVER EOF BOUNDARY	The subset specified extends over an end-of-file mark or a tape mark boundary.	Change the specified subset definition to a valid one and try the operation again.
123	SUBSET OPTION: THIS INPUT DEVICE DOES NOT BACKSPACE	The specified subset requires backspacing in the "from" file but the device for that file is not a disc or magnetic tape.	Check the MPE :FILE command associated with the "from" file and the back reference to it.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
124	READ ERROR IN FROMFILE AT RECORD *recnum*	An error occurred while spacing through the "from" file in search of the start of a subset.	Retry the operation.
125	SUBSET OPTION: NUMERIC SUBSET IS EMPTY	A subset specified by *starting- record-number*, *number-of- records*, and/or *last-record-number* does not exist or contains no data.	Change the specified subset definition to a valid one and try the operation again.
126	VERIFY OPTION: ERROR WHILE REWINDING FROMFILE	An error occurred while spacing backward to the beginning of the "from" file at the start of a verify operation.	Retry the operation.
127	VERIFY OPTION: ERROR WHILE REWINDING TOFILE	An error occurred while spacing backward to the beginning of the "to" file at the start of a verify operation.	Retry the operation.
128	EOF FOUND WHILE SPACING IN FROMFILE	An end-of-file mark was encountered while spacing through the "from" file in search of the start of a subset. This most often occurs when the "from" file is a blocked magnetic tape. For a blocked magnetic tape, the record numbers supplied in the SUBSET= parameter are used as block numbers.	Retry the operation specifying block numbers instead of record numbers. OR Reblock the tape so each block contains one record and then retry the operation.
129	EOF FOUND WHILE SPACING IN TOFILE	An end-of-file mark was encountered while spacing through the "to" file in search of the start of a subset during a compare or verify operation	*Compare operation*: The "from" and "to" files are not identical. Display the "to" file to determine what it actually contains. *Verify operation*: The copy operation was not performed correctly. Retry the operation.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
131	ERROR WHILE WRITING EOF TO TOFILE	An error occurred while writing an end-of-file mark in the "to" file.	Retry the operation
l32	VERIFY OPTION: ERROR WHILE SPACING IN THE FROMFILE	An error occurred while spacing through the "from" file during a verify operation.	Retry the operation
133	VERIFY OPTION: ERROR WHILE SPACING IN THE TOFILE	An error occurred while spacing through the "to" file during a verify operation.	Retry the operation
134	WARNING: FOUND EOF IN TOFILE	FCOPY has performed the specified operation but has filled the "to" file before completing the operation.	The "to" file was not large enough. Use the MPE :LISTF *filename*,2 command to determine the "to" file's size and then increase its size (using the MPE :PURGE and :BUILD commands) and retry the operation.
135	WRITE ERROR TO TOFILE	An error occurred while writing to the "to" file.	Retry the operation.
136	READ ERROR FROM TOFILE	An error occurred while reading from the "to" file during a compare or verify operation.	*Compare operation*: Retry the operation. If the error persists, you must try to recreate the "to" file. *Verify operation*: Retry the operation.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
137	WARNING: AN UNLABELLED TAPE OPERATION ENDS ON AN ERROR	An operation involving a magnetic tape "from" file was terminated by reading beyond the end of valid data rather than by sensing an end-of-file mark.	This is not an error. You can avoid this message by reading the "from" tape one file at a time and using the keyword SUBSET.
138	TITLE OPTION: TITLE TOO LONG	The title specified for the list function is longer than the 70 characters allowed or it extended over more than one line (record).	*In a job*: Correct the command and resubmit the job. *In a session*: Re-enter the command using the correct format.
139	DUMP OPTION: TOFILE RECSIZE NOT WITHIN LEGAL LIMIT	A file display was directed to an intermediate storage device with an incorrect record size. That record size must be $\leq$60 bytes (30 words).	Change the record size of the intermediate storage file (using the MPE :PURGE and :BUILD commands) so that it is within the allowed range and then retry the operation.
140	COMPARE OR VERIFY OPTION: OPERATION FAILS; DIFFERENT FIXED RECSIZES	The compare or verify operation was not attempted because the record sizes of the "to" and "from" files are not identical.	*Compare operation*: None. The compare operation revealed that the fixed record sizes of the two files are not identical. *Verify operation*: Change the record size of the "to" file (using the MPE :PURGE and :BUILD commands) so that it is the same as that of the "from" file and then retry the operation.
141	COMPARE BEGINS	The comparison phase of a verify operation has begun.	None.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
143	WARNING: FROMFILE IS EMPTY	The "from" file contained no data. Nothing was copied or compared.	None. You may have accidentally specified the wrong file as the "from" file.
144	NEW OPTION: FILE ALREADY EXISTS	The "to" file named for the new file function already exists in the specified (or implied) group and account.	Change the name of the "to" file and try the operation again.
145	BACKSPACE ERROR IN FROMFILE	An error occurred while spacing backward to the beginning of the "from" file or a subset within it.	Retry the operation.
200	WARNING: FROMFILE RECSIZE IS *number type*, TOFILE RECSIZE IS *number type*	The record sizes of the "from" and "to" files are not identical.	*In a job*: FCOPY performs the specified operation despite the conflict. *In a session*: You are given the choice whether or not to continue the operation. Note that if the "from" record size is larger than the "to" record size, the "from" records would be truncated. If the "to" record size is larger than the "from" record size, the content of the excess byte positions in the "to" records is unpredictable.
201	WARNING: FROMFILE IS ASCII, TOFILE IS BINARY or WARNING: FROMFILE IS BINARY, TOFILE IS ASCII	The data formats of the "from" and "to" files are not identical.	*In a job*: FCOPY performs the operation despite the conflict. *In a session*: You are given the choice whether or not to continue the operation.

Table A-5. FCOPY Warning and Error Messages (Continued)

CODE	MESSAGE	MEANING	ACTION
301	READ ERROR IN FROMFILE AT RECORD *recnum*	An error occurred while reading from the "from" file at the record number displayed (*recnum*).	Retry the operation. If the error persists, use the subset function to copy all of the file except the erroneous record.
302	VERIFY OPTION: RAN OUTOF VERIFY ERRORS AT FROMFILE RECORD *recnum*	The verify function was terminated because the specified maximum number of errors has been exceeded at the record number displayed (*recnum*).	Retry the operation specifying a larger *number-of-errors* parameter.
304	COMPARE OPTION: RAN OUT OF COMPARE ERRORS AT FROMFILE RECORD *recnum*	The compare function was terminated because the specified maximum number of errors has been exceeded at the record number displayed (*recnum*).	Retry the operation specifying a larger *number-of-errors* parameter
901	KSAM FROMFILE BOUNDARY (EOF OR BOF)	The beginning or end of the *from* file was reached during the copy operation.	
902	KSAM FROMFILE POSITIONING ERROR	Could not position to desired place.	Try again.
903	ERROR; WRONG CONDITIONS FOR OPENING NEW KSAM FILE	From file is not a KSAM file, or NO KSAM was specified.	Create a KSAM file before running FCOPY and copy to that file.

B

KSAM/3000 INTERNAL STRUCTURE AND TECHNIQUES

OVERVIEW

KSAM files can be used efficiently without any knowledge of how the files are structured or how file blocking and size is determined. The default values provided for file capacity, key blocking, number of key entries, and so forth are effective in many applications. This appendix provides the sophisticated programming staff with information on how KSAM files are structured, how disc space is allocated to a KSAM file, and how memory space is allocated for the Extra Data Segments used when a KSAM file is modified or accessed. Such information may be useful for improving performance based on the particular application.

KSAM FILE STRUCTURE

A KSAM/3000 file is two physical files: a data file and a key file. The data file portion of a KSAM file contains all the data in the file and contains nothing but the data. Data records are written to the data file in the order in which they are received from a program. (The last record added is always written to the end of the file.) This chronological order is not necessarily in sequence by key value. At the time the file is opened, you can specify that records must be written in primary key sequence, but the default mode is to write records in any order.

The key file portion of a KSAM file contains the key entries that maintain the sequence of the data records. As a data record is written to the data file, a key entry is added for each key defined for the file, and the sequential connections between key entries maintained. This means that if there is a primary key and two alternate keys, three key entries are added with each new data record, and three sets of pointers are updated to reflect the new key sequence of each key.

The structure of the data file is like that of any MPE file. Data records may be fixed or variable in length. If fixed, each record is the size specified when the file is created (default size is equivalent to one 128-word disc sector). If variable, the actual size of each record is included in the record itself, and the maximum size of any record is used to determine the blocking. By default, data records are blocked one record per block.

The structure of the key file is more complex. The key file is organized so that locating a particular key requires the least number of accesses. For this purpose, the key files are organized in a particular structure known as a "B-Tree".[1]

Described in *"Organization and Maintenance of Large Ordered Indexes"* Bayer and McCreight, Acta Informatica, Springer Verlag,1972, pp 173-189.

B-tree structure has two main advantages:

- The number of file accesses is limited to the number of levels in the tree. If there are two levels, no more than two reads of the key file are needed to locate a particular key.

- The key file is balanced. This means that each level pointer associated with a particular key value points to approximately as many higher key values as lower key values at the next level of the tree.

B-tree structure in general is discussed below, followed by a discussion of how KSAM key files use this structure.

B-TREE STRUCTURE

In a B-tree, there is always one root level block that points to blocks at a lower level. At the lowest level, the blocks are called leaves and they do not reference another level. In a two-level structure (see Figure B-1), the blocks at the second level all "leaves". If the tree has more than two levels, intermediate blocks (nodes or branches) are referenced by a higher level and themselves reference a lower level. Unless this lower level is a leaf, it also references a lower level. This continues until the lowest (leaf) level is reached.

The notion of higher and lower level does not refer to the key values, The root block key values are always central and point to blocks with lower values and blocks with higher values. Thus if there are two entries at the root or a branch level, there will be three pointers to the next level: one for key values less than the first key value, one for key values less than the second key value but greater than the first, and one for key values greater than the second.

Within each block, values are stored in ascending order. Although not all blocks are filled with values, each block in a tree is the same size. Figure B-1 illustrates a simple 2-level tree with one root block and three leaf blocks. The root is a single block and each leaf is a block of the same size. (This example uses the KSAM minimum key block size consisting of four key entries per block.)

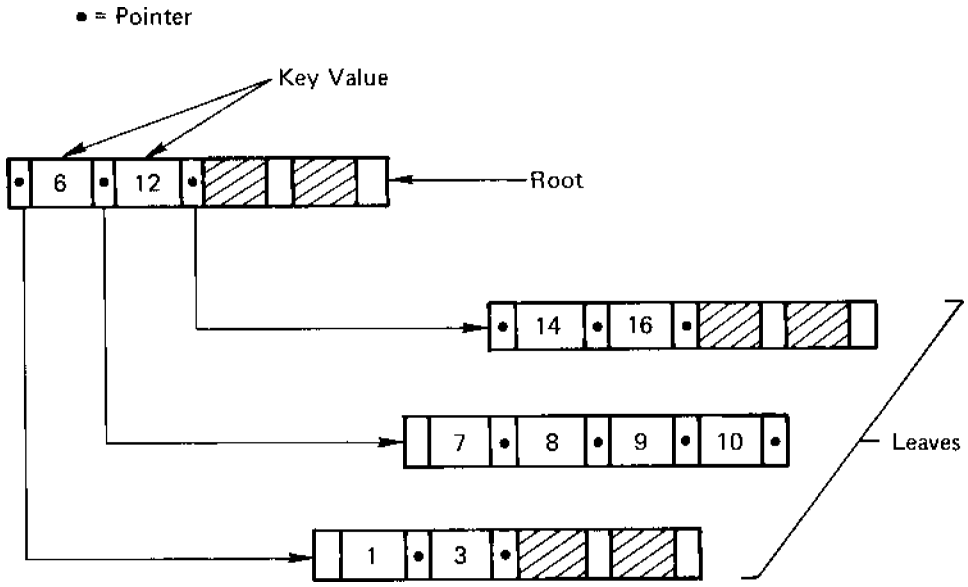


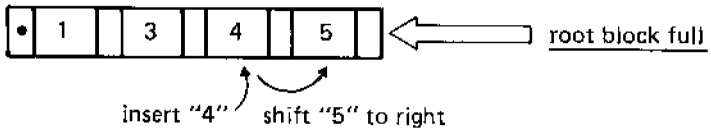
Figure B-1. Two-Level B-Tree Structure

ADDING OR DELETING KEYS.

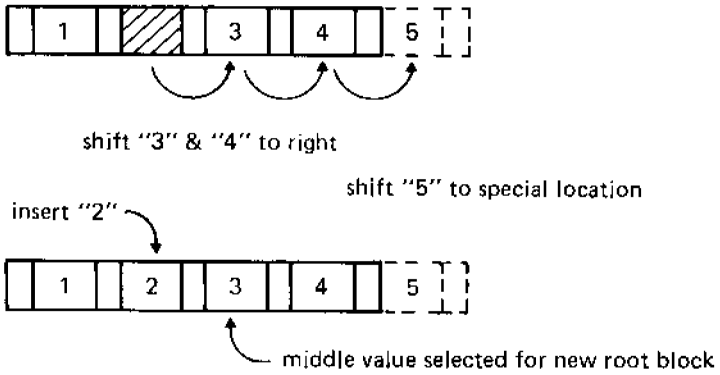
When a key block is full and new keys are added, the root level block may need to be split, causing a new root block to be introduced and adding a new level to the tree. This process is illustrated in Figure B-2 where the addition of one new key to a partially filled block does not affect the tree structure, but the addition of a second key to the full block causes the block to split.

Again, this example assumes the minimum key block size for the sake of simplicity. Note that all key file maintenance is performed in the KSAM extra data segment where space is allocated for one more key than the key block size. This allows the addition of a key to a "full" block. Before the block is read back into the key file, it is split so that the key block size is maintained.

1. Insert key value "4":



2. Insert key value "2" into full block:



3. Split block in order to maintain block size:

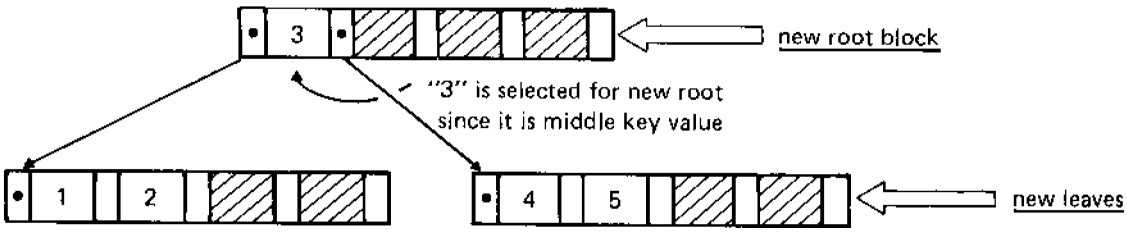


Figure B-2. Split Causes New Level in Tree

When the root block and all the leaves are full, another split becomes necessary. Figure B-3 illustrates a split caused when a new key is added to a full two-level tree structure, forcing it to a three-level structure.

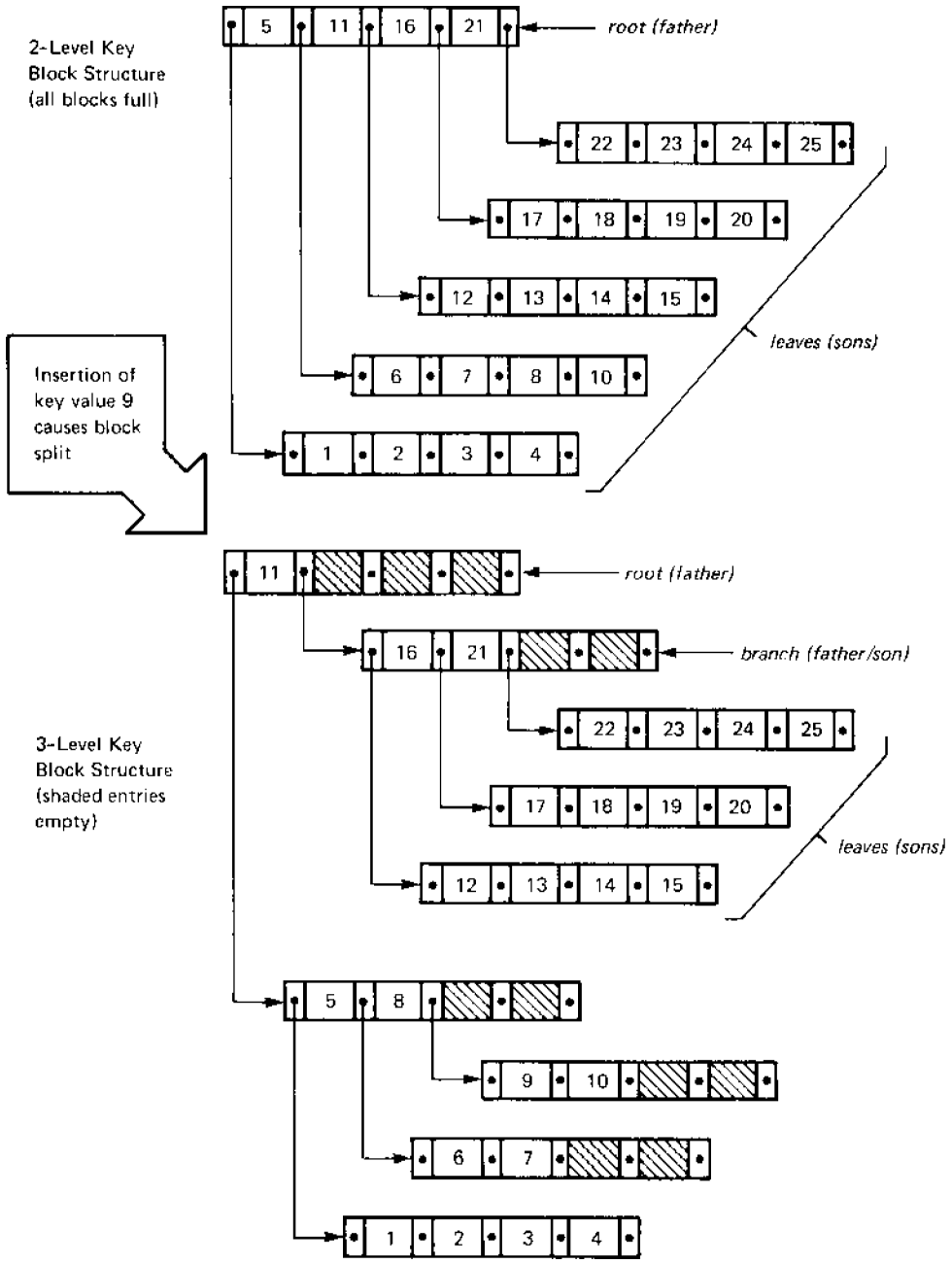


Figure B-3. Tree Growth from Two to Three Levels

Note that key blocks must always be defined with an even number of keys. As a result, when a key is added to a full block, there will be a middle value to form a block at a new level. This maintains the balance essential to B-tree structure.

As records are deleted from the data file, two blocks at the same level (brothers) may be merged into one block. If sufficient records are deleted, the root block may be merged into a higher level, thereby contracting the number of levels in the key structure.

KSAM KEY FILE STRUCTURE

A KSAM key file consists of three types of information:

- Control—contains general control information such as the KSAM filename, and the number of keys defined for the file,
- Key descriptor—contains general key information for each key such as the starting location in the data record of the key field, and the location in the key file of the root key entry.
- Key entries—Each key entry contains information about a key associated with a data record. This information consists of:
  - the key value
  - a pointer into a data record in the data file with the same key value.
  - pointers into other records within the key file.

The control and key descriptor information is contained in two blocks (physical records) at the beginning of each file. Regardless of the number of keys in the file, each block is 128 words (1 sector) long. Thus, every key file is preceded by two sectors of control and key descriptor information.

The key entries are also organized into blocks of a fixed size. However, the exact number of blocks and the size of each block is based on a variety of factors, such as the key size, the number of keys in the file, the number of key values for each key, the key blocking factor, and so forth. (Calculation of key block size is discussed later in this section.) These key entry blocks are organized into the Btree structure discussed above. A separate key structure is maintained for each key defined for the file, Thus there may be up to 16 separate tree structures in a single KSAM file.

Refer to Figure B-4 for a simplilied diagram of a KSAM key file with two keys each organized into a two level tree structure. For a detailed description of the three types of block, refer to Figure B-5 and Figure B-6.

CONTROL BLOCK.

This 128-word block contains information on the data file associated with the key file, and keeps track of the number and type of access to the key file. It also specifies the number of keys (primary and alternate) defined for the KSAM file. The name of the data file and the number of keys are essential for associating the key file with the data file. The number of keys determines how many entries are in the Key Descriptor Block. (Refer to Figure B-5.)

KEY DESCRIPTOR BLOCK.

This 128-word block contains one 8-word entry for each key defined for the KSAM file. The first entry describes the primary key, the next entry describes the first alternate key (if there is one), and each subsequent entry describes any additional alternate keys. The first word of each entry points to the root block for that key; another important item is the location of the key in the data file record. (Refer to Figure B-5.)

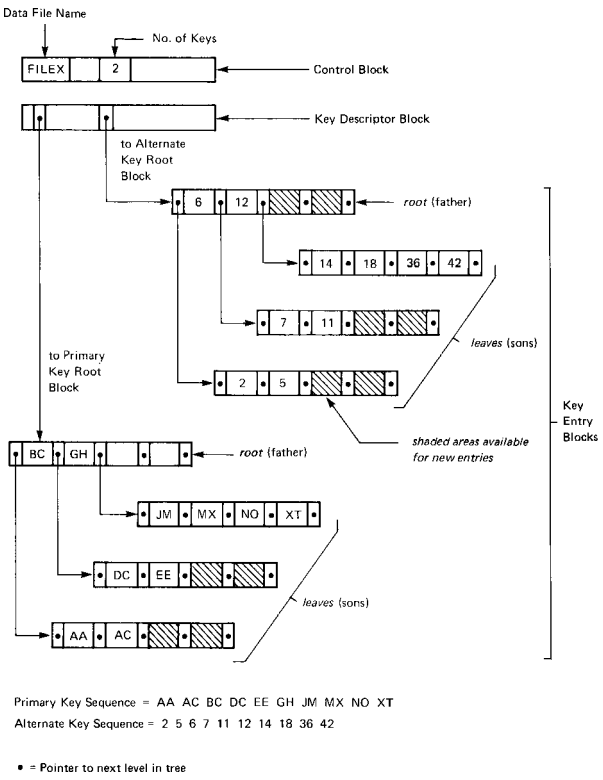


Figure B-4. KSAM Key File Structure With Two Keys

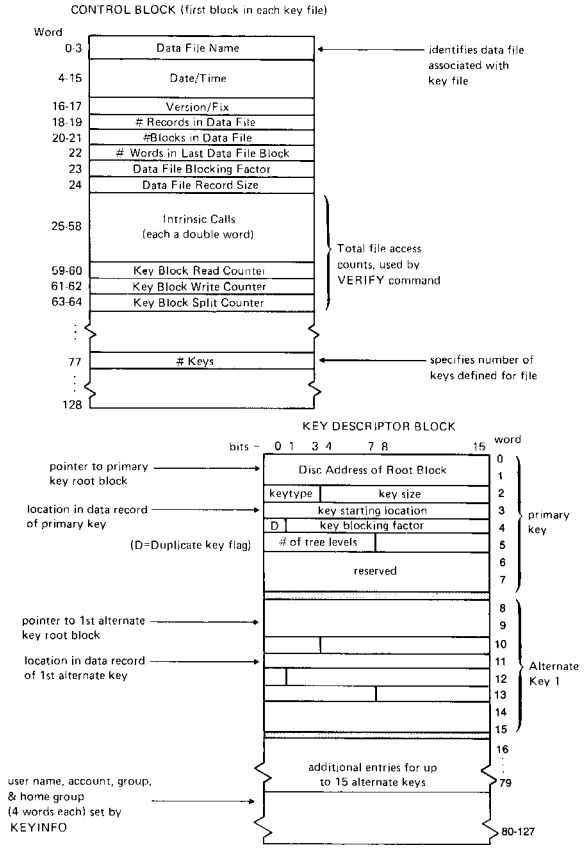


Figure B-5. Control Block and Key Descriptor Block

KEY ENTRY BLOCKS.

Each block in the key file contains, in addition to the key values, pointers that link the key blocks to each other and pointers that link each key value to an associated data record. Preceding these entries, the first item in every key block is the address of the block on disc; the next item is the number of keys in the key block.

All key block access for search and modification is performed in the KSAM extra data segment. The disc address in each key block insures that the block is returned to its correct location on disc from the extra data segment.

Figure B-6 illustrates the general layout of all key entry blocks. Each key value is followed by a pointer to a data record and a pointer to the block at the next level with higher key values. The first, pointer in each block points to a block at the next level with lower key values. These pointers are set to zero for key blocks that have no next level (the leaves on a tree structure).

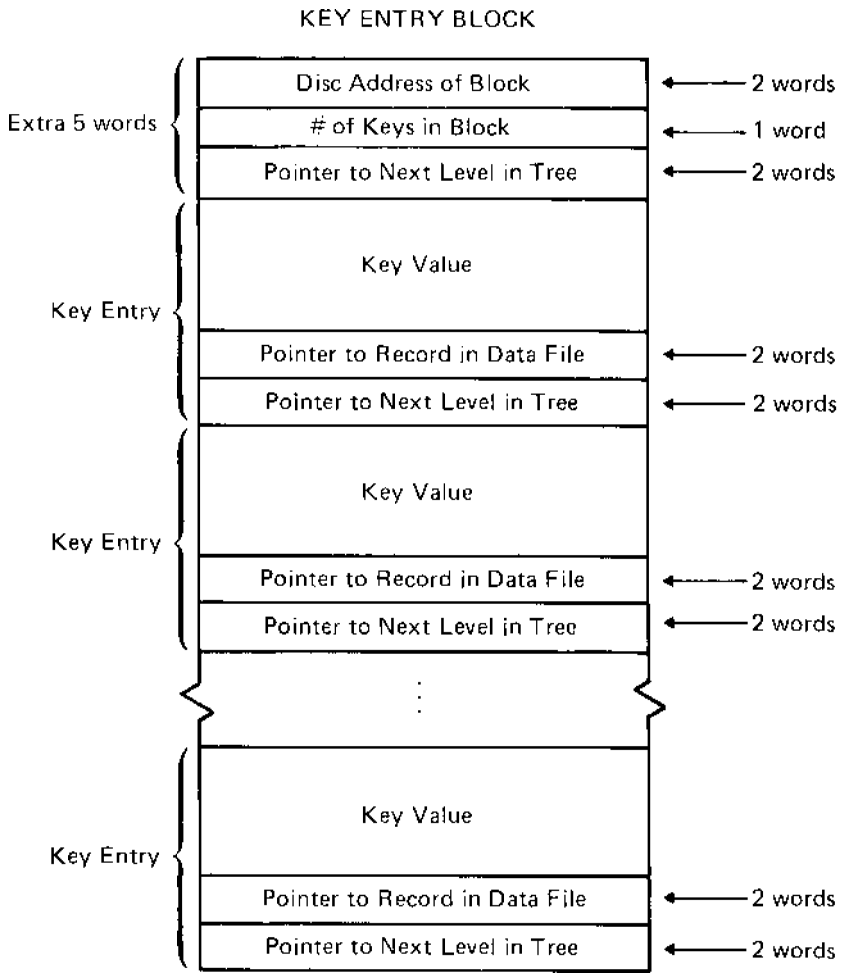


Figure B-6. Key Entry Block Structure

RELATION OF KEY TO DATA FILE

The purpose of the KSAM key file is to maintain the order of data records in the data file. In order to maintain sequential order for each key, the keys blocks are connected through pointers. In addition to these pointers, each key entry must also contain a pointer linking the key value to the data record containing the corresponding key value.

When the KSAM file is created, each key is defined by its starting location in the data record, its length, and its type. The location is specified as the character position where the key value starts; the length is the maximum number of characters used by the key value; its type is the type of the value such as, an integer, a character string, or a double-word integer.

Thus, if the primary key is defined as a character string that starts in character position 3 and is 20 characters long, then KSAM expects that each data record will contain such a value in that location. Whatever is placed in the defined location is treated as the primary key and determines the order in which data records are sequenced.

The order in which records are physically written to the date file is called *chronological sequence*. This sequence may or may not also be a *key sequence*. If the records were written to the file in primary key sequence, then this sequence and the chronological sequence are the same. If there is an alternate key for the file, however, it is very unlikely that alternate key sequence is the same as the chronological sequence.

Note Key sequence in KSAM files is always in ascending order by key value.

Refer to Figure B-7 for a simplified diagram of the relation between the primary keys in the key file and the associated data records in the data file. (A similar diagram could be set up for the alternate key.) The diagram shows the pointer in each key entry pointing directly to a record in the data file.

When a data record is to be located by key value, the root block for the appropriate key is searched first, using a binary search method. If the key is in the root block, the search is over. If it is not, the key value is between two root block values or it is less than the lowest value or greater than the highest. Using the pointer in the appropriate location, a block at the next level is located. This block is then searched for the selected key. Again, if the key is found, the search is over. If the key is not found at this level, the appropriate pointer to the next level is used and the search continues.

When the selected key value is found, the pointer to the data file associated with that key value is used to locate the record in the data file.

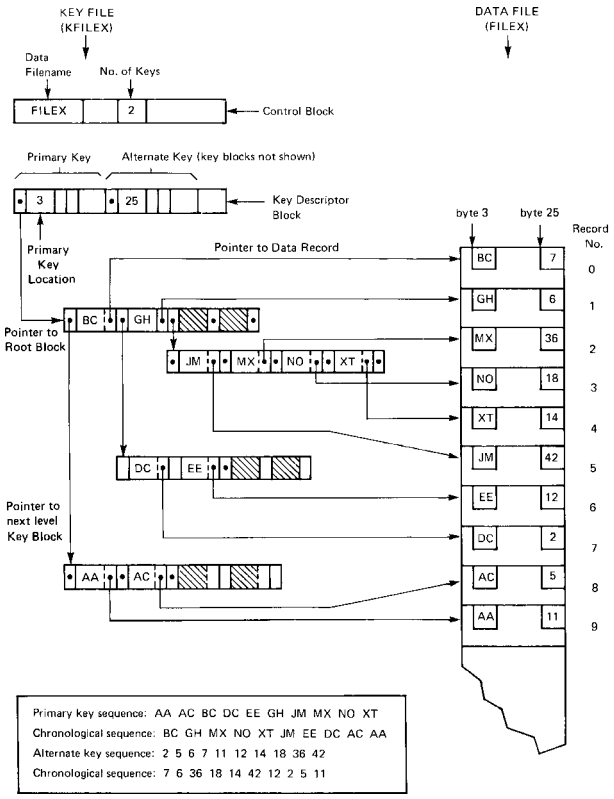


Figure B-7. Data File/Key File Relation

KSAM FILE SIZE

The size of the data file is calculated from the maximum number of data records times the size of each record (for fixed length records). For variable length records, it is calculated from the maximum number of data blocks times the size of each block. By default, a KSAM data file contains 1024 records (or blocks) in which each record (or block) is 1024 words long. This default size fits each block into eight disc sectors (each sector is 128 words long), and results in a data file of 8192 sectors.

Calculation of key file size is more complex. It is based on the total number of keys in the file (primary and alternate), the size of each key entry (including overhead), the expected number of data records specified when the file is built, plus space to allow for block splitting when the number of key entries increases.

The number of key entries per key is usually exactly the same as the number of data records expected. By default, KSAM uses the maximum number of data records specified, or the default value of 1024 records. This number is multiplied by each key in addition to the primary key to arrive at the total number of key entries in the file.

The size of each key entry and the number of key entries per block (the blocking factor) is used to determine key block size. Since all blocks in the key file must be the same size, KSAM adjusts the blocking factor so that all keys, regardless of entry size, use the same block size. Also, this blocking factor may be adjusted so that disc sector space is not wasted. (A block always starts on a sector boundary.) By default, the blocking factor is adjusted so that a block size of 1024 words is used for all key blocks for all keys in the file.

Because of the nature of the B-tree structure, enough room must be left in the key file so that the file can be increased in a balanced manner. When block splitting occurs as a result of adding new key values, up to half of each key block may have empty slots. To allow for such expansion, the key file size is calculated, and then doubled.

The following discussion describes exactly how KSAM calculates the key block size, and then the total key file size. These calculations are useful primarily if you do not use default values for the key blocking factor and for the number of key entries. In such a case, they may help you determine the most effective block size and file size for your application.

KEY BLOCK SIZE

Key block size can affect the complexity of the tree structure and this complexity can affect access time. In order to understand the relation between block size and access time, consider the following general rules:

- *The larger the block, the less often it has to split and the fewer the splits, the fewer levels to the tree.*
- *The more levels to the tree, the more mass storage retrieval time is needed to locate a particular y value.*

From this it would follow that in order to reduce access time, you should define large blocks. This is true only up to a point. Depending on the total number of key values expected in the file, a large block size may result in a great deal of unused space in each block. Also, the blocking factor may result in unused disc space since all blocks must start on sector boundaries.

KSAM provides a default blocking factor that produces a block with 1024(1K) words. This size has proved to be efficient for many files. You may, however, override this default blocking by specifying a value in the *keyblocking* parameter of the ;KEY= option in the BUILD command, or in word 19 of the FOPEN *ksamparam* parameter. Note that any blocking factor you specify is a minimum value since KSAM may increase the blocking factor so that the least amount of disc space is wasted.

After creating a KSAM file, you can use the VERIFY command of KSAMUTIL to determine the number of levels needed by the KSAM file. The VERIFY listing will also tell you the actual blocking factor used in creating the file so you can find out whether your specified blocking factor has been increased.

CALCULATING KEY BLOCK SIZE.

Key block size is based on a number of factors:

- The key size is bytes (KS)
- The key entry size in words (ES).
- The number of key entries per block, the blocking factor (BF).

Once the block size is determined, the number of sectors needed to hold one block is calculated. If this value (NB) wastes sector space, KSAM adjusts the blocking factor to produce a block size that uses the least number of sectors by filling each sector as completely as possible. Note that when KSAM uses the default block size of 1024, it calculates a blocking factor by the same method.

The following steps show how KSAM determines block size based on a specified minimum blocking factor.

Note The notation |_ _| means round down the result of the enclosed algorithm to the next whole number; | | means round it up.

---

1. Calculate the entry size (ES) in words from the key size (KS) in bytes, and then add two words for each pointer in the key entry (see Figure B-6). KSAM uses the following algorithm to perform this calculation:

   ```
   ES = |_(KS+I)/2_| + 4
   ```

2. If the blocking factor (BF) was specified as an odd number, KSAM issues an error message. Otherwise, it uses the specified blocking factor to continue the calculation of block size.

3. Determine blocksize (BS) by multiplying the key entry size by the blocking factor and adding 5 words. (The five words are for the three words of control information at the beginning of each block, plus two words for the final pointer in the block. See Figure B-6). KSAM uses the algorithm:

   ```
   BS = (ES x BF) + 5
   ```

   Since blocks always start on sector boundaries, this calculated block size may leave a lot of unused sector space. The following steps show how KSAM determines the most efficient block size and, if this size differs from the size calculated from the specified blocking factor, how KSAM adjusts this blocking factor upwards to produce the optimum block size.

4. Determine the number of sectors required to hold the block at its calculated size. If the result is not a whole number, round it up to the nearest whole sector. KSAM determines the number of sectors per block (NB) as follows:

```
NB = | BS/128 |
```

5. Multiply the number of sectors per block by 128 to determine the optimum block size:

```
BS = NB x 128
```

6. If the optimum block size calculated in step 5 differs from the block size calculated in step 3, or if the default block size 1024 is used, KSAM adjusts the blocking factor to one that produces the optimum block size. It uses the following algorithm to determine the number of key entries that fit in the block and, if this is an odd number, reduces it by one. (Blocking factor must be an even number.)

```
BF = F | (|_(BS-5)/ES _| −1)/2 |x 2
```

KEY FILE SIZE

KSAM uses the blocking factor and the number of sectors per block to detenrmine the maximum file size in sectors to allocate to the key file. In addition, KSAM needs to know the maximum number of key entries to expect, and the number of keys (primary and alternate) defined for the key file at creation.

The maximum number of key entries for each key may be specified in the *numentries* parameter of the KEYENTRIES= option of the BUILD command, or in the *ksamparam* parameter of FOPEN. However, this file limit is usually based on the maximum number of data records. This value is specited in the *numrec* parameter of the DISC= option of the BUILD command, or in the *filesize* parameter of FOPEN. If not specified in either of these places, KSAM assumes a default file limit of 1024 key entries.

Since the number of records in the data file can be used to calculate the maximum number of keys for only one key, each additional key defined for the file causes the file sizes to be summed.

When key file size has been calculated, KSAM uses this value to allocate that number of sectors on disc to the key file whenever the file is opened.

Key file size is based on the following factors:

- The number of key entries per block, or the blocking factor (BF).
- The number of blocks per sector (NB).
- The maximum number of key entries for one key (FL).
- The number of keys defined for the key file.

KSAM uses the following formula to determine key file size in sectors for a file with one key:

```
FBS = (| FL/BF |x 2) x NB
```

This formula is derived through the following steps:

1. The maximum number of key entries (FL) is divided by the number of key entries per block (BF) to find the number of blocks in the file. If not a whole number, it is rounded up to include all blocks.

2. The resulting number of blocks is multiplied by 2. This doubling of the number of key blocks is done to allow room for expansion when the number of levels in a key file expands due to block splitting.
3. Finally, the number of blocks is multiplied by the number of blocks per sector (NB) to find the total number of sectors needed to contain all key entries.

Note The file size (FS) calculated by the above algorithm does not include the two sectors required for control information.

If the file has only one key, add 2 to the file size (FS+2) to get the total file size. The value 2 represents the two sectors at the beginning of each key file containing control and key descriptor information.

If the key file has multiple keys, then the optimum block size of each key must be determined. The largest block size is then selected as the standard block size for all keys in the file (KSAM does not allow variable-length key blocks). Since the block size of some keys has changed, the blocking factor (BF) must be recalculated for these keys using the algorithm:

```
BF = | (|_(BS-5)/ES_|−1)/2 |x 2
```

A separate file size for each key is then calculated based on their various blocking factors. The total key file size is equal to the sum of all these file size (FS) values plus 2 for the two control sectors.

Figure B-8 summarizes the steps KSAM uses to calculate file size for one key. Figure B-9 shows how KSAM calculates key file size for a file with two keys. Each key file size (FS) is calculated separately, and then the blocking factor and file size of the key with the smaller block size is recalculated.

For a file with one key, KSAM simply adds 2 sectors to the file size (FS) calculated for the single key.

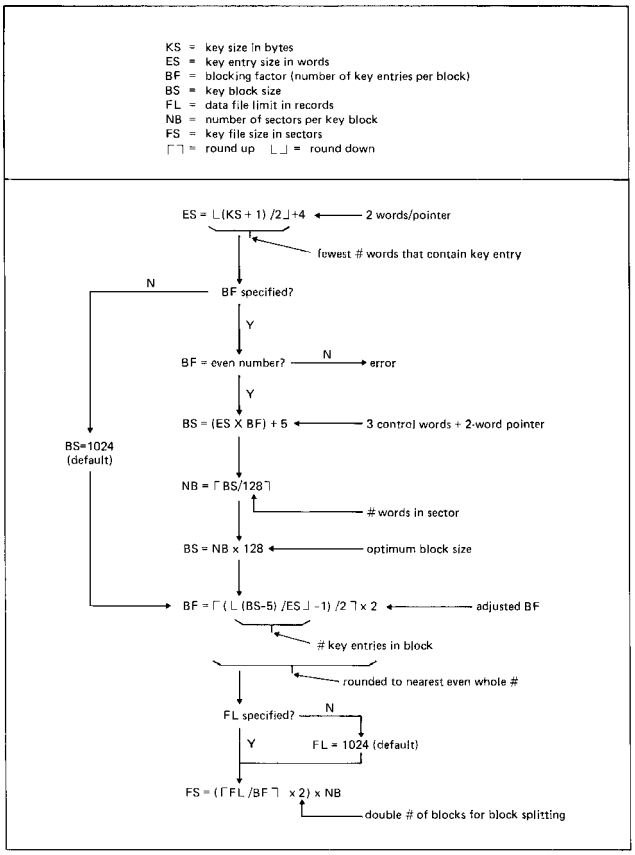


Figure B-8. Formula to Determine File Space per Key

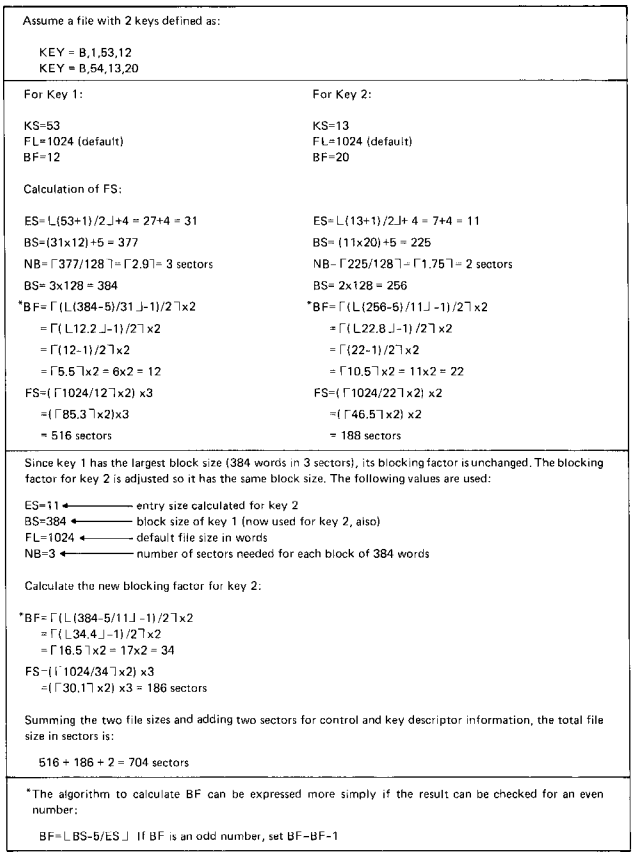


Figure B-9. Calculation of Total Key File Size with Two Keys

KSAM EXTRA DATA SEGMENTS

Another factor that may affect performance when KSAM files are used, is the number and size of KSAM extra data segments. An extra data segment (XDS) is an area in memory used as a buffer during KSAM file access. Each extra data segment contains:

- Statistical information on file use (listed by VERIFY command);
- Control Block and Key Descriptor Block data from the first two sectors of each KSAM key file;
- A Working Storage area large enough to hold a data record and two key entries;
- A data block buffer large enough to hold a block from the data file;
- At least one, and up to 20, key block buffers each large enough to hold one key block.

When the key file is searched for a particular data record, the root block and lower level blocks, as needed, are moved to key block buffers in the extra data segment. Key entries are compared in the working storage area. When the data block is located, it is moved to the data block buffer of the extra data segment, and when the particular data record is located, it is moved to the working storage area.

Since each open KSAM file is allocated an extra data segment and each extra data segment can be up to 32K words long (32,767 words), KSAM files can use a lot of memory. When there is not enough room in memory for all the extra data segments, they must be swapped between memory and disc as needed. This swapping can slow access to KSAM files.

In order to minimize swapping, you can reduce the number of KSAM files by combining several files into one file. This automatically reduces the number of extra data segments, and it can be a very effective way to improve performance, particularly when files are shared by a number of users. (Refer to Number of Extra Data Segments, below.)

Decreasing the overall size of the extra data segment may reduce swapping of extra data segments. However, reducing the number of key block buffers in the extra data segment may increase swapping of key blocks between the key file and the extra segment during a file search. By default, KSAM allocates key block buffers according to a formula that takes into account the type of access for which the file is opened, the number of levels in the key file structure, and the number of alternate keys in the file. Since this formula (see Table B-1) keeps the extra data segment size as small as is compatible with efficiency, the default number of key block buffers should be used except in special cases. (For details, refer to Extra Data Segment Size later in this section.)

NUMBER OF EXTRA DATA SEGMENTS

KSAM assigns an extra data segment to each KSAM file opened by an active process. Since more than one process can use the same file during shared access, one file may require a number of extra data segments. Thus, the number of extra data segments depends both on the number of KSAM files and the number of users concurrently using the file. (Refer to Figure B-10.)

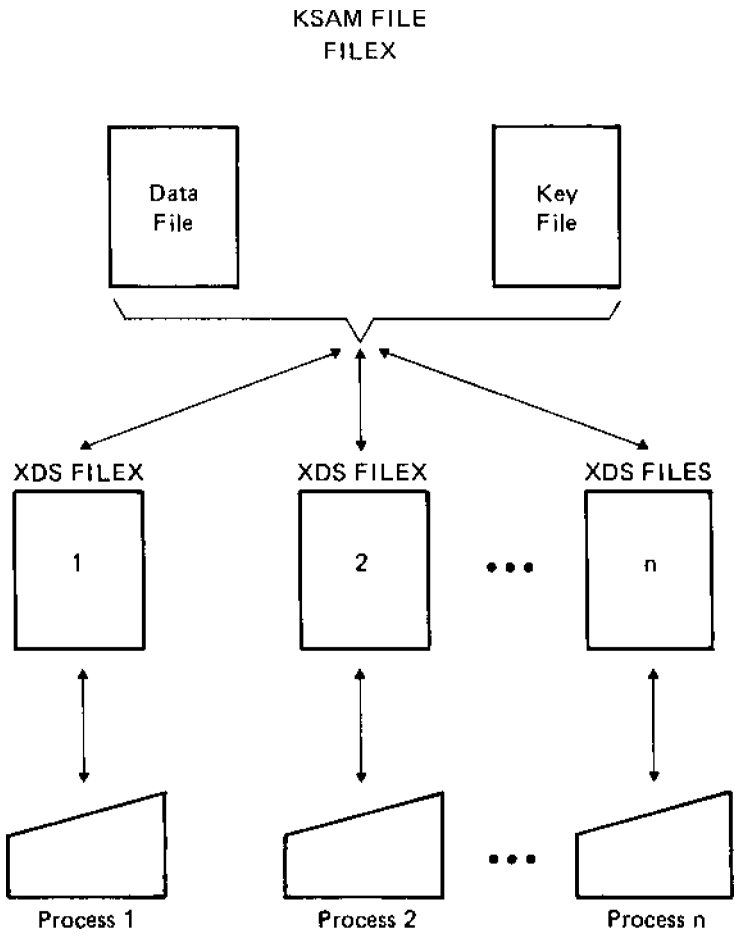


Figure B-10. Extra Data Segments for Shared Access

EXTRA DATA SEGMENT SIZE

The size of each extra data segment associated with an open KSAM file is determined by the number of key block buffers it contains, the size of each key block buffer, the size of the data block buffer, and to a lesser extent, the key entry size and the data record size. (Refer to Figure B-11.)

Initially (when a file is opened), 12K words are allocated to the extra data segment. If less actual space is needed, the extra space is not used, but remains in virtual memory. If more is needed, the original extra data segment is released and a new extra data segment is allocated with the actual size needed.

The maximum size of any extra data segment is 32K words. The actual size is calculated from:

- The total size of the overhead statistics and working storage area;
- The size of the data block buffer;
- The size of each key block buffer and the number of buffers allocated.

The overhead statistics and working storage area is approximately 1-1/2K bytes long depending on variables such as the key entry size and the data entry size. The data block buffer size is based on the size of each data block in the data file. Each key block buffer must be large enough to contain all the key entries in a key block plus one key entry used when new keys are added to a full block (as described earlier, see Figure B-2).

The default key block size is 2K bytes (1024 words) and the maximum size of the key block buffer is 4K bytes (2048 words). If a key block is larger than 4K bytes, KSAM reduces the block size so that no block is larger than will fit in an extra data segment key buffer. Thus, the main variable in extra data segment size is the number of key block buffers.

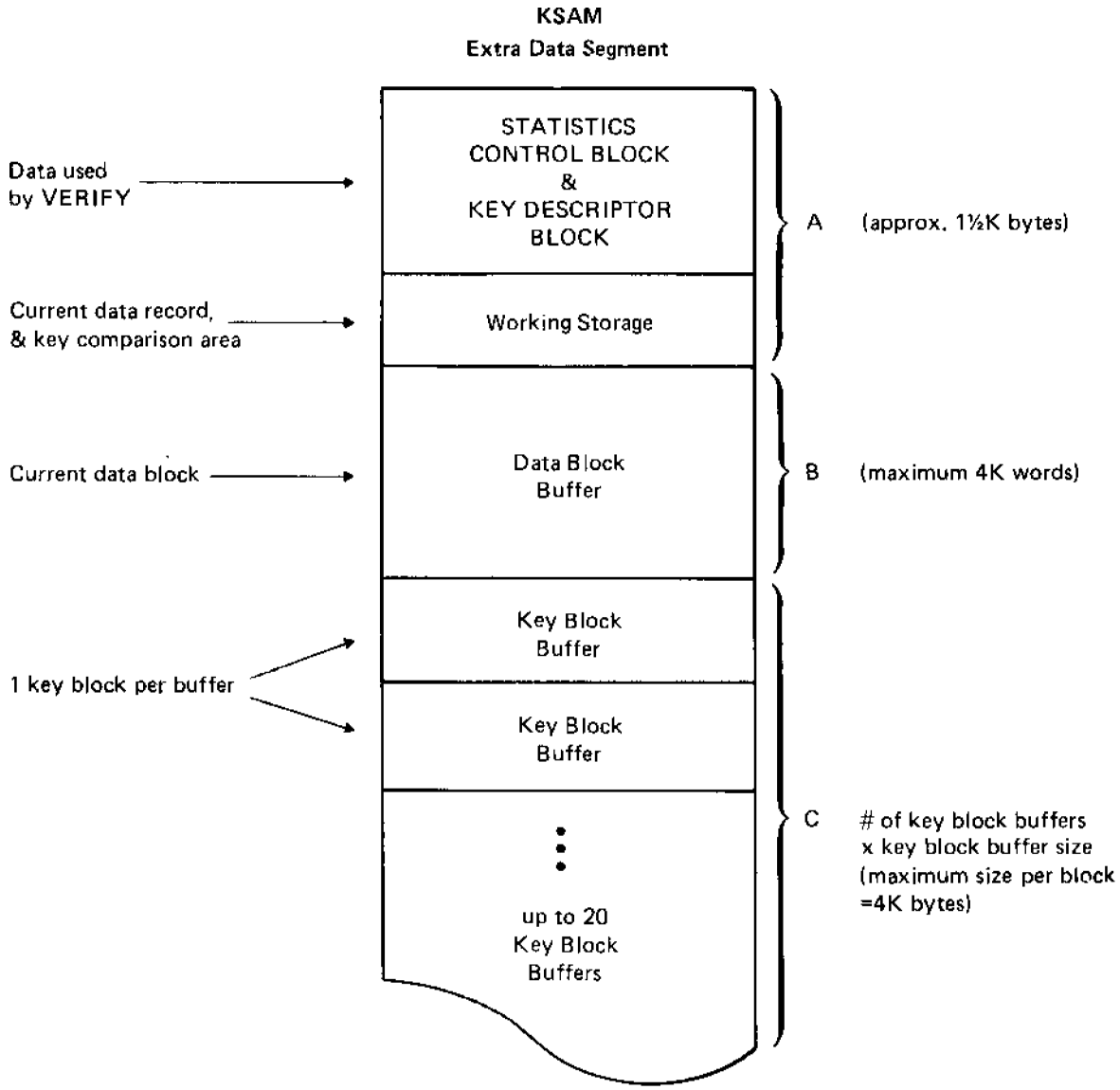


Figure B-11. KSAM Extra Data Segment

NUMBER OF KEY BLOCK BUFFERS.

The number of key block buffers depends on the type of access for which the file is opened, the number of keys in the file, and the number of levels in the tree structure for each key. (Refer to Table B-1 for details.) The least number of buffers is allocated for read only access, unless the primary key has many levels in its structure. More buffers are usually required for write only, read/write, or update access. The number of buffers for read only access increases with the number of levels used by the key, but is never less than one. The number of buffers for write only access increases with the number of alternate keys in the file, but is never less than six. The number of buffers for all other types of access increases with the number of alternate keys and with the number of levels for each key, but is never less than four.

Unless you specify a particular number of key block buffers, KSAM allocates buffers in the extra data segment according to the file characteristics as shown in Table B-1.

Table B-1. Number of Key Block Buffers Assigned by Default

Access Type	Buffers Assigned
Read Only Access	1 buffer per level in key with most levels (minimum of 1 buffer up to 20 buffers)
Write Only Access	3 buffers per primary key + 3 buffers per alternate key + 3 buffers (minimum of 3 buffers up to 20 buffers)
Other Access (Read/Write or Update)	1 buffer per level in primary key structure + 1 buffer per level in alternate key structure + 3 buffers (minimum of 3 buffers up to a maximum of 20 buffers)

Note that you can determine the number of levels per key with the KSAMUTIL command, VERIFY.

For example, if the file is opened for read only, and the only key needs two levels, two key block buffers are allocated. If the file is opened for write only, and there is one alternate key in the file, nine key block buffers are allocated. If this same file is opened for update access, the primary key uses two levels, and the alternate uses three, a total of eight buffers is allocated.

If you want to override the number of key block buffers assigned by default, you can use the MPE :FILE command before opening the file, or set the *numbuffers* parameter of FOPEN when you open the file programmatically.

The file equation is specified as follows:

`:FILE` *filename*`; DEV=,,`*#buffers*

The KSAM extra data segment will be allocated space for as many key block buffers as you specify, up to a maximum of 20. (Note that the third DEV= parameter is interpreted as the number of key block buffers only when the file name is a KSAM file; for standard MPE files, this parameter indicates the number of list copies of the file.)

Another way to reduce the number of key block buffers is to use fewer alternate keys, or to adjust the blocking factor so that the key file structure uses fewer levels. Either of these methods is effective when the file is written to or updated more than it is simply read.

Note that when you are loading a KSAM file with large amounts of data, you should increase the number of key buffers. The more key buffers in the extra data segment, the more likely it is that, as new data is added, locations for the new key values will be found in memory. This cuts down on disc access and can significantly reduce the time it takes to load the file.

For example, if you are reloading a KSAM file after a system failure, you should use the :FILE command to increase the number of buffers to maximum of 20 buffers. Then, after the file is loaded, you can allow it to revert to the default number of buffers established by KSAM for the particular file.

EXTRA DATA SEGMENTS WITH SHARED ACCESS

The extra data segment allocated to each open file acts as a control block for that file. The extra data segment contains not only the current data block and the current key block buffers, but also the latest control information for the file. This information includes the logical and chronological record pointers that indicate the current record being accessed. Because the current pointer position is not in a "common block", when several programs open the same file, each can alter the key file structure by adding or deleting records so that the pointers set by other programs may point to the wrong record without those other programs being aware of it.

To make sure that the latest pointer position is stored with the file rather than in the separate extra data segments, programs that share the same KSAM file must use a locking scheme. Whenever a program locks a KSAM file, the control information is transferred from the file to the extra data segment; and when a program unlocks the file, the contents of the extra data segment is written back to the file. Thus, each program should lock a KSAM file before executing any procedure that positions a record pointer (pointer-independent procedures), and not unlock the file until all procedures that depend on this pointer position (pointer-dependent procedures) have completed execution. This is true regardless of whether the pointer is chronological (points to a record in the data file) or is logical (points to a key in the key file). Both types of pointer are maintained in the extra data segment for the open file.

Table B-2 lists all the procedures that affect or are affected by the record pointers.

Table B-2. Pointer Dependence

Pointer-Independent Procedures	Pointer Type	Pointer-Dependent Procedures	Pointer Type
FFINDBYKEY CKSTART BKSTART	Logical	FREAD CKREAD BKREAD	Logical
FFINDN	Logical	FSPACE	Logical
FREADBYKEY CKREADBYKEY BKREADBYKEY	Logical	FREMOVE CKDELETE BKDELETE	Logical
FWRITE CKWRITE BKWRITE	Logical	FUPDATE CKREWRITE BKREWRITE	Logical
FPOINT	Chronological	FREADC	Chronological
FREADDIR	Chronological*	FREADC	Chronological

*Each procedure that sets the logical pointer also sets the chronological pointer; but only FPOINT sets the logical pointer as well as the chronological pointer.

The pointer-independent calls position the pointer regardless of its current position. Pointerpendent calls, on the other hand, must know to which record the pointer is currently positioned in order to operate correctly.

All the procedures listed in Table B-2 affect the pointer in some way. In order to use these procedures correctly, it is important to understand how each moves the pointer, whether it positions the pointer directly or advances it from its current position.

In general, when access to the file is random, the pointer is positioned directly. For example, a call to FFINDBYKEY (or CKSTART or BKSTART) positions the logical pointer to a particular key in the key file based on a key value specified in the call; and a call to FPOINT positions the chronological pointer to a particular record determined by its chronological record number.

When access to a file is sequential or the file is being modified, pointer positioning is not direct but is relative to its previous position. Depending on the sequence in which procedures are executed, the pointer may or may not be advanced to the next record in key or chronological sequence. Internally, a flag is used to indicate whether or not to advance the pointer. This flag, the "Do Not Advance" flag, is set to FALSE if the pointer is to be advanced sequentially, to TRUE if it is not to be advanced. Some procedures never test the flag; these are, in general, the pointer-independent procedures that set the pointer directly. Other procedures test the flag and advance the pointer if the flag is FALSE; generally, these are procedures that read the file or position the pointer sequentially. Table B-3 summarizes when the pointer is set or advanced. (Note that only SPL procedures are listed; check Table B-2 for the equivalent BASIC or COBOL procedures.)

Table B-3. Record Pointer Summary

Beginning of Call		Procedure	End of Call	
Check DNA Flag	Record Pointer		Record Pointer	Sets DNA Flag
–	–	FFINDBYKEY	Position	TRUE
–	–	FFINDN	Position	TRUE
–	Position	FREADBYKEY	–	FALSE
–	–	FWRITE	Advance	TRUE
If TRUE If FALSE	– Advance	FREAD	–	FALSE
If TRUE If FALSE	– Advance	FSPACE	Position	TRUE
–	–	FREMOVE	Advance	TRUE
–	–	FUPDATE (key value changed)	– Advance	FALSE TRUE
–	–	FPOINT	Position	TRUE
–	Position	FREADDIR	–	FALSE
If TRUE If FALSE	– Advance	FREADC	–	FALSE

Advance: Move logical pointer to next record in key sequence or move chronological pointer to next record in chronological sequence.
Position: Set pointer to record specified in call.

For example, if you call FREADBYKEY, it positions the pointer to a specified key value. After the call, the logical pointer remains positioned to this key and the Do Not Advance flag is set to FALSE. If the next call is to FREAD, FSPACE, or FREADC, then the pointer is advanced to the next key in key sequence before these procedures perform their other functions. Thus, after FREADBYKEY, a call toFREAD will read the next record, not reread the same record, and a call to FSPACE will move the pointer relative to the record following the record just read.

C

ASCII CHARACTER SET IN COLLATING SEQUENCE

In the collating sequence for ASCII characters, unlike EBCDIC, numbers precede letters.

Table C-1. ASCII Characters in Sequence

DECIMAL VALUE	CONTROL	GRAPHIC	COMMENTS
0	NUL	@^c	Null
1	SOH	A^c	Start of heading
2	STX	B^c	Start of text
3	ETX	C^c	End of text
4	EOT	D^c	End of transmission
5	ENQ	E^c	Enquiry
6	ACK	F^c	Acknowledge
7	BEL	G^c	Bell
8	BS	H^c	Backspace
9	HT	I^c	Horizontal tabualation
10	LF	J^c	Line feed
11	VT	K^c	Vertical tabulation
12	FF	L^c	Form feed
13	CR	M^c	Carriage return
14	SO	N^c	Shift out
15	SI	O^c	Shift in
16	DLE	P^c	Data link escape
17	DC1	Q^c	Device control 1 (X-ON)
18	DC2	R^c	Device control 2
19	DC3	S^c	Device control 3 (X-OFF)
20	DC4	T^c	Device control 4
21	NAK	U^c	Negative acknowledge
22	SYN	V^c	Synchronous idle
23	ETB	W^c	End of transmission block
24	CAN	X^c	Cancel
25	EM	Y^c	End of medium
26	SUB	Z^c	Substitute
27	ESC|[^c	Escape	
28	FS	\^c	File Separator
29	GS	]^c	Group Separator
30	RS	^^c	Record separator

Table C-1. ASCII Characters in Sequence (cont.)

DECIMAL VALUE	CONTROL	GRAPHIC	COMMENTS
31	US	_	Unit separator
32	SP	‘	Space (blank)
33	!	a	Exclamation point
34	”	b	Quotation mark
35	#	c	Number sign
36	$	d	Dollar sign
37	%	e	Percent sign
38	&	f	Ampersand
39	’	g	Apostrophe
40	(	h	Left parenthesis
41	)	i	Right parenthesis
42	*	j	Asterisk
43	+	k	Plus sign
44	,	l	Comma
45	-	m	Minus sign (hyphen)
46	.	n	Decimal point (period)
47	/	o	Slash
48	0	p	Zero
49	1	q	One
50	2	r	Two
51	3	s	Three
52	4	t	Four
53	5	u	Five
54	6	v	Six
55	7	w	Seven
56	8	x	Eight
57	9	y	Nine
58	:	z	Colon
59	;	{	Semicolon
60	<	|	Less than

Table C-1. ASCII Characters in Sequence (cont.)

DECIMAL VALUE	CONTROL	GRAPHIC	COMMENTS
61	=	}[c]	Equals sign
62	>	~[c]	Greater than
63	?	DEL[c]	Question mark
64	@		Commercial at
65	A		Uppercase A
66	B		Uppercase B
67	C		Uppercase C
68	D		Uppercase D
69	E		Uppercase E
70	F		Uppercase F
71	G		Uppercase G
72	H		Uppercase H
73	I		Uppercase I
74	J		Uppercase J
75	K		Uppercase K
76	L		Uppercase L
77	M		Uppercase M
78	N		Uppercase N
79	O		Uppercase O
80	P		Uppercase P
81	Q		Uppercase Q
82	R		Uppercase R
83	S		Uppercase S
84	T		Uppercase T
85	U		Uppercase U
86	V		Uppercase V
87	W		Uppercase W
88	X		Uppercase X
89	Y		Uppercase Y
90	Z		Uppercase Z

Table C-1. ASCII Characters in Sequence (cont.)

DECIMAL VALUE	CONTROL	GRAPHIC	COMMENTS
91	[		Opening bracket
92	\		Reverse slant
93	]		Closing bracket
94	^		Circumflex
95	_		Underscore
96	`		Grave accent
97	a		Lowercase a
98	b		Lowercase b
99	c		Lowercase c
100	d		Lowercase d
101	e		Lowercase e
102	f		Lowercase f
103	g		Lowercase g
104	h		Lowercase h
105	i		Lowercase i
106	j		Lowercase j
107	k		Lowercase k
108	l		Lowercase l
109	m		Lowercase m
110	n		Lowercase n
111	o		Lowercase o
112	p		Lowercase p
113	q		Lowercase q
114	r		Lowercase r
115	s		Lowercase s
116	t		Lowercase t
117	u		Lowercase u
118	v		Lowercase v
119	w		Lowercase w
120	x		Lowercase x
121	y		Lowercase y
122	z		Lowercase z
123	{		Left brace
124	|	Vertical line	
125	}		Right brace
126	~		Tilde
127	DEL		Delete

D

CONVERSION TO KSAM FILES

In order to convert from your existing files to KSAM files, you may want to take advantage of utility programs provided with KSAM/ 3000. If your files are serially accessible, you can use the KSAMUTIL command BUILD to create a KSAM file and then copy your files to the new file with FCOPY. Another method only converts HP INDEX files. INDEX is the new name for RSAM (or R'ISAM) files. This method uses the program RTOKSAM. Finally, if neither of these methods is useful, you can write your own special purpose conversion program.

USING KSAMUTIL AND FCOPY

This conversion method can be used for any file that is serially accessible. First you create a KSAM/3000 file using the BUILD command of KSAMUTIL. At this time you can define your file with any legitimate specification of the BUILD command. Once the file is built (created), you can run FCOPY to copy your existing file to the newly created file. No special FCOPY commands are needed. You simply specify your existing file as the FROM= file and the newly created KSAM file as the TO= file. All connections between the data file and the key file are made automatically. (Refer to section II of this manual for a discussion of both the KSAMUTIL BUILD command and FCOPY as applied to KSAM/3000 files.)

USING RTOKSAM

The RTOKSAM program will create a KSAM/3000 file from an existing INDEX file. The KSAM file will have the same key structure as the INDEX file. Any number of INDEX files can be converted to KSAM/3000 files in one RTOKSAM run provided that you have sufficient disc space within your group and account for all the files.

Program RTOKSAM is run as follows:

```
        :RUN RTOKSAM.PUB.SYS.
HP32208.A.0.00 INDEX TO KSAM CONVERTER
ENTER INDEX KEY, KSAM DATA, AND KSAM KEY FILE NAMES
>indexkey,ksamdata,ksamkey
```

The names of the INDEX key file, the KSAM data file, and the KSAM key file must be entered in that order. Only the INDEX key file already exists; a new KSAM/3000 file wll be created with the specified names.

After converting the INDEX file to the KSAM file, the program continues to prompt you for additional file names for conversion. When you wish to stop processing, simply press the *carriage return* key in response to the greater than (>) prompt. Or, if you are in a job, enter an :EOD record.

Record numbering in INDEX files always starts with record number 1. The KSAM file created by the RTOKSAM conversion program will also have record number starting with 1. Note that this is not the standard KSAM file default. Key blocking, on the other hand, does follow the KSAM file defaWt. That is, the number of keys per block is determined by KSAM so that each key block has 1024 (1K) words.

If errors occur during execution of RTOKSAM, the following error messages may be displayed:

```
INPUT/OUTPUT ERROR ON $STDIN/$STDLIST
COMMAND TOO LONG
DUPLICATED FILE NAME
INSUFFICIENT PARAMETERS
INDEX OPEN ERROR                        (detail line follows message)
INDEX FREADLABEL ERROR                  (detail line follows message)
UNABLE TO BUILD KSAM FILE               (detail line follows message)
KSAM FILE WRITE ERROR                   (detail line follows message)
INDEX FILE READ ERROR                   (detail line follows message)
```

The detail line that follows certain of the RTOKSAM messages explains in more detail the input/output error that occurred.

The normal MPE security provisions for files apply when the INDEX file is specified in this program. The KSAM file that is created must be within the same log on group.

Note It is good practice to make a back-up copy of the INDEX file on off-line storage such as magnetic tape before running RTOKSAM to copy the file to a KSAM file. This allows you to purge the INDEX file once it is copied.

E

RECOVERY FROM SYSTEM FAILURE

OVERVIEW

If the system fails when a KSAM file is open for any type of access except read-only, the file cannot be reopened until it has been recovered. In such a circumstance, any attempt to reopen the file causes the following message to be issued:

```
#192 -- SYSTEM FAILURE OCCURRED WHILE THE KSAM FILE WAS OPENED
```

The file is easily recovered in most cases by running KSAMUTIL.PUB.SYS and then requesting KEYINFO. This command resets any incorrect end-of-file marks and deletes any key values that point to non-existent data records. If key values are missing or are out of sequence, the keys cannot be recovered by KEYINFO and, in this case, the file must be reloaded. (You can refer to section II for a discussion of KEYINFO; also an example of file recovery and reloading is provided later in this appendix,) If you want to examine the file statistics, you can run the VERIFY command of KSAMUTIL using the NOCHECK option. (If KEYINFO does not complete execution successfully, then the KSAM file must be reloaded.)

For most purposes, this is all you need to know in order to recover a file when a system failure prevents you from opening it. This appendix provides internal details that explain why recovery is necessary and what KEYINFO does in order to recover. It is intended primarily for the sophisticated programming staff.

END-OF-FILE ON KSAM FILES

The first step in understanding what KEYINFO does and why it is needed, is to understand how KSAM end-of-files are set and maintained. Each of the two files that comprise a KSAM file (the data Ele and the key file) has two end-of-file marks: an MPE end-of-file and a KSAM internal end-of-file. Thus, there are four end-of-files to consider. The main characteristics of each of these end-of-files are shown below:

DATA FILE

MPE End-Of-File:

- Number of records in fixed-length record file (or number of blocks in variable-length record file).
- Stored in system file label of data file.
- Recorded on disc when file is *closed* (or when an SPL procedure calls FCONTROL with control code 6) or when a new extent is allocated.
- Used by FCOPY with NOKSAM option (KSAM file is treated as an MPE file).
- Displayed by LISTF,2:

```
LISTF DATAFIL,2
ACCOUNT=  MORRIS       GROUP=  JOAN

FILENAME  CODE   ------------LOGICAL RECORD-----------  ----SPACE----
                    SIZE  TYP       EOF      LIMIT R/B  SECTORS #X MX

DATAFIL   KSAM       38B   FA        20         20   1       23  8  8
                                     ^
                                     |---- MPE end-of-file for data file
```

KSAM End-of-File:

- Address of next available logical record in the data file.
- Stored in control block of key file.
- Recorded on disc when file is *unlocked* or *closed* (or when an SPL procedure calls FCONTROL with control code 2 or 6).
- Used by FCOPY when file is opened as a KSAM file (KEY-option)
- Displayed by VERIFY command (option 1) of KSAMUTIL:

```
:RUN KSAMUTIL.PUB.SYS


HP32208A.2.3 MON, APR 23, 1979, 1:11 PM KSAMUTIL VERSION:A.2.0
VERIFY DATAFIL


WHICH (1=FILE INFO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL,)?1

DATAFIL.JOAN.MOPRIS        CREATOR=JOAN
FOPTIONS(004005)=KSAM, :FILE, NOCCTL, F, FILENAME, ASCII, PERM
AOPTIONS(000400)=DEFAULT, NOBUF, DEFAULT, NO FLOCK, NO MR, IN
RECSIZE:SUB:TYP:LDNUM:DRT:UN.:  CODE:LOGICAL PTR: END OF FILE:FILE LIMIT
    -38:  3:  0:    3:  5:  0:    0:          0:          20:        20
                                                          |
```

KSAM end-of-file for data file----------

```
LOG. COUNT:PHYS. COUNT:BLK SZ:EXT SZ:NR EXT: LABELS:LDN :    DISCARD:
         1:          1:   -38:     3:     8:       1:    3:00000010135:
```

Since this is a file that closed successfully, the two end-of-files coincide.

KEY FILE

MPE End-of-File:

- File limit - Number of records (sectors) allocated to file at time of creation
- Stored in system file label.
- Displayed by LISTF,2:

```
:LISTF KEYFIL,2
ACCOUNT=  MORRIS      GROUP=  JOAN


FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----
                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX


KEYFIL    KSAMK   128W  FB          50         50   1       30  5  8
                                     |
                                     |
                       MPE end-of-file on key file
                          (set to file limit)
```

KSAM End-of-File

- Address of 1st record in next available key block.
- Stored in Control Block of key file.
- Recorded on disc when file is *unlocked* or *closed* (or when SPL procedure calls FCONTROL with control codes 2 or 6).
- Used by FCOPY and KSAM procedures.
- Displayed by VERIFY command, option 3, of KSAMUTIL:

```
WHICH (1=FILE INTO, 2=KSAM PARAMETERS, 3=KSAM CONTROL, 4=ALL)?3

DATA FILE = DATAFIL     VERSION= A.2.1
KEY CREATED=292/''78 10:19: 7.4      KEY ACCESS= 113/''79 13:11:45.8
KEY CHANGED= 93/''79 14:18: 7.6      COUNT START=292/''78 10:19:53.6
DATA RECS  =         20 DATA BLOCKS=         19 END BLK WDS=              19
DATA BLK SZ=         19 DATA REC SZ=         38 AZCESSORS=                 0
FOPEN                 2 FREAD                 0 FCLOSE                     2
FREADDIR              0 FREADC                0 FREADBYKEY                 0
FREMOVE               0 FSPACE               57 FFINDBYKEY                 0
FGETINFO              3 FGETKEYINFO           1 FREADLABEL                 0
FWRITELABEL           0 FCHECK                0 FFINDN                     3
FWRITE               20 FUPDATE               0 FPOINT                     0
FLOCK                 0 FUNLOCK               0 FCONTRDL                   0
FSETMODE              0
KEYBLK READ           7 KEYBLK WROTE          0 KEYBLK SPLIT               0
KEY FILE EOF         26 FREE KEY HD           0 SYSTEM FAILURE             0
                        |
              KSAM internal end-of-file on key file
MIN PRIME            11 MAX PRIME             5 RESET DATE             3/''79
DATA FIXED         TRUE DATA B/F              1 TOTAL KEYS                 3
FIRST RECNUM          0 MIN RECSIZE          38
```

Since the MPE end-of-file is set to the file limit and the KSAM internal end-of-file to the next available key block, these values never coincide until the key file is full.

END-OF-FILE AND THE EXTRA DATA SEGMENT

As described in appendix B, each open KSAM file uses an extra data segment (XDS) to hold the control information for that particular open file. The extra data segment also contains a data block buffer into which records are read from the file and from which records are written. Finally, the extra data segment keeps key block buffers to hold key entries affected by the data records being accessed. The control block in each extra data segment also maintains the most up-to-date KSAM end-of-file markers for each open file.

Whenever a KSAM file is opened, the KSAM end-of-files for the data and key files are moved (with all other information from the key file control block) to the control block of the extra data segment for that file. When the file is closed or unlocked, the control block is written back to disc. (Refer to Figure E-1for a diagram illustrating the end-of-file markers and their relation to an extra data segment.)

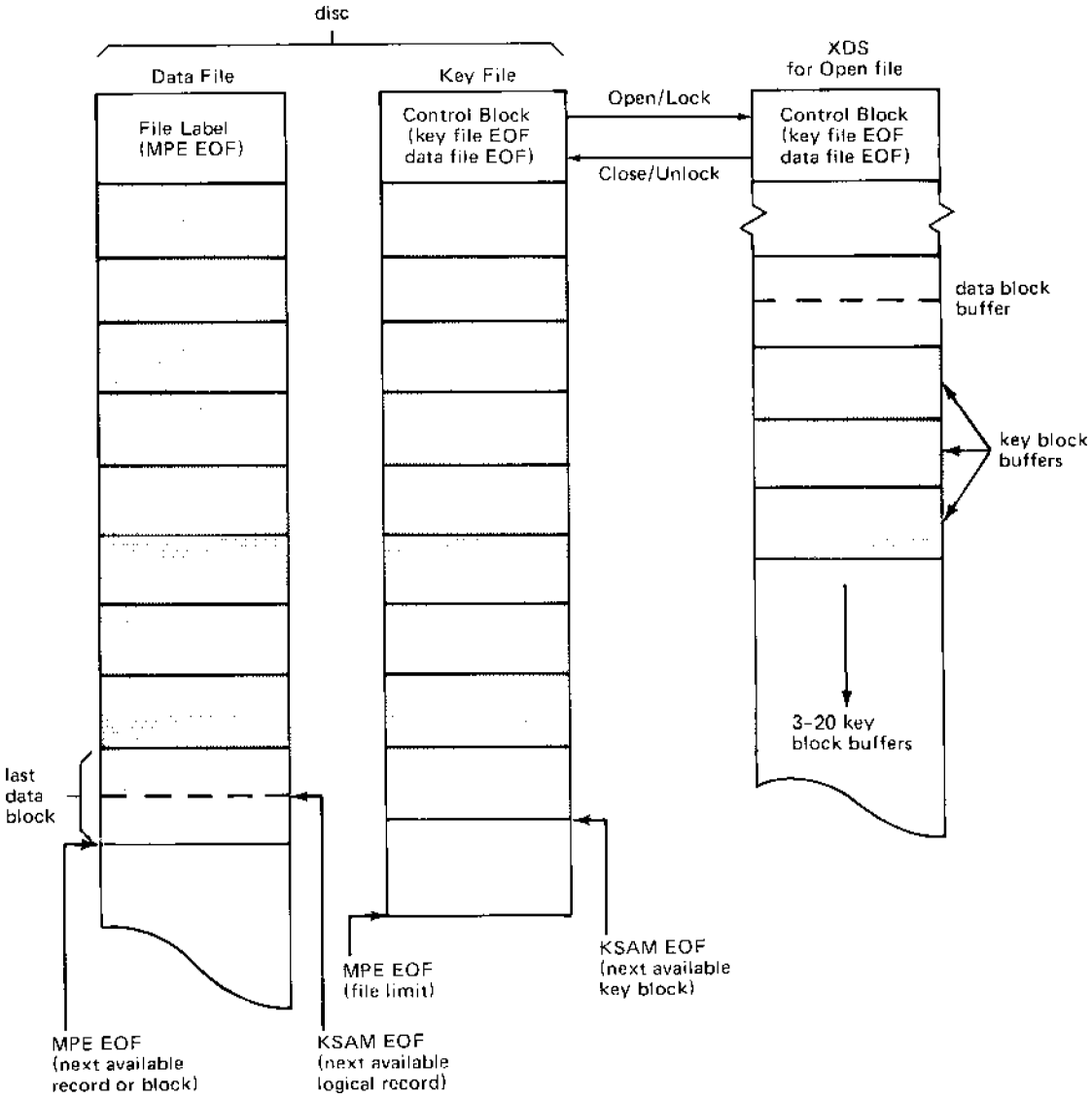


Figure E-1. KSAM File and an Extra Data Segment

NORMAL OPERATION - FILE IS CLOSED

During normal operation, if a new record is written to the file by any user, the record is written in the data block buffer of the extra data segment and the key entry for the record is inserted into the key block buffer where it belongs. (Refer to appendix B for a discussion of how new key entries are added.) The appropriate key block buffer is brought into the extra data segment automatically. Then, whenever the data block or key block buffers are full or new blocks must be read into the extra data segment, the key and data blocks are written back to disc. But the control block from the extra data segment is not written to disc until the file is closed (or is unlocked, or FCONTROL with code 2 or 6 is called).

Before considering what happens in case of a system failure, let's look at the normal steps taken when the file is closed:

1. Key block buffers are written to the key file
2. Data block buffer is written to the data file (and, if a new extent is allocated, the MPE end-of-file is written to the data file system label).
3. Control block with the KSAM end-of-file marks is written to the key file.
4. MPE end-of-file mark is written to the data file system label.

When a file is unlocked, the first three steps shown above are taken (except the MPE end of file is not written). FCONTROL with control code 6 performs all four steps, and control code 2 performs the first three steps.

SYSTEM FAILURE - FILE IS OPEN

If the system fails when a KSAM file is open, the extent of the damage to the file depends on when the failure occurred and whether the file was being modified. If all users opened the file for readonly, then the file is undamaged and can be reopened. If a user had just unlocked the file and no other user has modified it, the MPE end-of-file may need to be reset but otherwise, the file is undamaged. But if the file was being modified, then the extent of the damage depends on whether any of the steps listed above had been completed and, if so, which ones.

In the simplest case, all the steps except step 4 have been performed. This means that the KSAM end-of-file is up-to-date, but the MPE end-of-file is still at its previous position, In the most complex case, caused by records being deleted, data records remain in the data file for which there are no corresponding key entries, (error number 175).

SITUATIONS IN WHICH RECOVERY IS REQUIRED

Whenever the file cannot be reopened (error #192 is issued), you must run KEYINFO to recover the file. The following four cases are typical of the reasons for file damage. In each case, the suggested action is discussed.

1. Records were being added to the data file when the system failed. The KSAM end-of-file for the data and key files are current, but the MPE end-of-file precedes the KSAM end-of-file (steps 1-3 completed).

   Solution: Run KEYINFO to reset the MPE end-of-file. You can then run the KSAMUTIL command, VERIFY, to determine where the current KSAM end-of-file for the data file is positioned, and then run the MPE command :LISTF 2 to compare the MPE end-of-file. If you run VERIFY before running KEYINFO, use the NOCHECK option so the file can be opened.

2. Records being added to the KSAM file when the system failed were not written to the data file, but some key entries for the new records had been written to the key file (key blocks written to key file because buffer space in XDS was needed). This means that the key file contains key values pointing to records not in the data file.

   Solution: Run KEYINFO to delete the key values that point to records that were not written to the data file.

3. Records being added to the KSAM file when the system failed caused a key block split. As a result, the key blocks are written, but the new internal KSAM end-of-file for the key file has not been transferred to disc. This places certain key values past the old KSAM key file end-of-file.

   Solution: Run KEYINFO to reset the key file end-of-file to the correct location following the existing key values. It still may have to delete any key values pointing to records past the data file end-of-file.

4. Records were being deleted when the system failed. Some key block buffers have been written to disc, but the data block buffer has not. Since some key entries were deleted from the file on disc, but the deleted records remain, key values appear to be missing.

   Solution: You must run KEYINFO to reset the crash flag so the file can be reopened. When key values are missing, KEYINFO cannot fully recover from the file damage and issues the following message:

Warning **THERE ARE SOME RECORD(S) WITH KEY VALUE(S) MISSING THE KSAM FILE HAS TO BE RELOADED**

   To reload the KSAM file, use FCOPY to copy the file to a new KSAM file. As it copies the data records, it writes new key entries for each data record. Only in this way can missing key values be recovered. (Refer to the discussion of Reloading a KSAM File later in this section.)

If you want to determine how many key values are missing and the file has more than one key, you can compare the number of values in each B-Tree as listed by KEYINFO. These values should be identical. When there is only one key in the file, you can use FCOPY to determine the number of non-deleted records in the batch file. The number of key values for any key in

the file should exactly match the number of valid data records. The FCOPY command to determine this value is:

```
>FROM=filename;TO=$NULL;KEY=0
```

If your file is very large, using this FCOPY command can be time consuming and you may prefer to reload the file without checking the number of missing keys.

EXAMPLE OF FILE RECOVERY

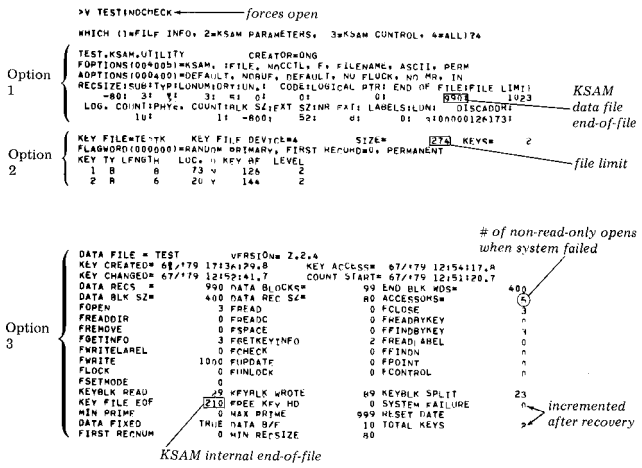
Suppose you try to open file TEST and receive error message #192:

```
SYSTEM FAILURE OCCURRED WHILE KSAM FILE WAS OPENED
```

In order to have the most information about the file, first run KSAMUTIL and request VERIFY to list all the file information.

```
HP32208Y.2.4. THU, MAR 8, 1979, 12:53 PM KSAMUTIL VERSION:A.3.0
>V TEST

WHICH (1=FILE INFO, 2=KSAM PARAMETERS,  3=KSAM CONTROL, 4=ALL)?4

SYSTEM FAILURE OCCURRED WHILE THE KSAM FILE WAS OPENED  (1068)
                                                        |
          KSAMUTIL error message--------------------------
```

Just like any other program, the VERIFY command cannot open the damaged file. So try again using the NOCHECK option that allows such a file to be opened for read-only:



The next step is to run LISTF,2 to see where the MPE end-of-file is positioned. Note that you can request the MPE command without exiting from KSAMUTIL.

```
>:LISTF TEST,2
ACCOUNT= UTILITY       GROUP= KSAM
FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----
                  SIZ   TYP        EOF      LIMIT R/B  SECTORS #X MX

TEST      KSAM    80B   FA         900       1023  10      416  8  8
                                     |
                                  MPE end-of-file on data file
```

LISTF shows the MPE end-of-file after the first 900 records, whereas option 1 of VERIFY showed the KSAM end-of-file after 990 data records. This is a discrepancy of 90 records. These records actually exist. You only have to run KEYINFO to reset the MPE end-of-file. When you run KEYINFO, however, you may find that there are other problems.

```
>KI TEST
RECOVERY BEGINS

DATA FILE EOF DAMAGED

DATA FILE MPE EOF HAS BEEN RESET TO KSAM EOF <----- MPE end-of-file recovered
```

After resetting the MPE end-of-file for the data file, KEYINFO continues. It next displays information on the two keys in the file TESTK.

```
--------- INFO FOR KEY               1 ---------
# OF LEVELS OF B-TREE                        2
# OF KEY BLOCKS                             16
# OF SECTORS PER KEY BLOCK                   8 <-------KSAM end-of-file should
# OF KEYS IN ROOT KEY BLOCK                 14         be at least 218
# OF KEYS IN B-TREE                       1000 <----|     |
% OF KEY BLOCK UTILIZATION                  52.1    |     |
THE LARGEST KEY BLOCK ADDRESS               210 <----------|
                                                    |
--------- INFO FOR KEY               2 ---------     |
# OF LEVELS OF B-TREE                        2         # of key entries in two
# OF KEY BLOCKS                             11      |  keys do not match
# OF SECTORS PER KEY BLOCK                   8      |  each other; do not match
# OF KEYS IN ROOT KEY BLOCK                  9      |  # of data records
# OF KEYS IN B-TREE                        997 <----|
% OF KEY BLOCK UTILIZATION                  68.6
THE LARGEST KEY BLOCK ADDRESS               202

WARNING: THERE ARE SOME RECORD(S) WITH KEY VALUE(S) MISSING
         OR KEY VALUE(S) POINTING TO DATA RECORD BEYOND EOF
KEY FILE EOF(INTERNAL) DAMAGED
KEY FILE (INTERlNAL)EOF Has BFEN RESET <---- End-of-file moved forward
                                             so all key blocks are included
```

Looking at the key information displayed for the two keys, the first thing to check is where the actual end of file should be. The largest key block address for key 1 is 210 and each block requires 8 records, therefore the key file end-of-file should be at least 218. If you look back to option 3 of the VERIFY display, it is listed as 210. Since this is not the real end-of-file, KEYINFO resets the KSAM internal end-of-file to the actual end of file (see VERIFY display, below).

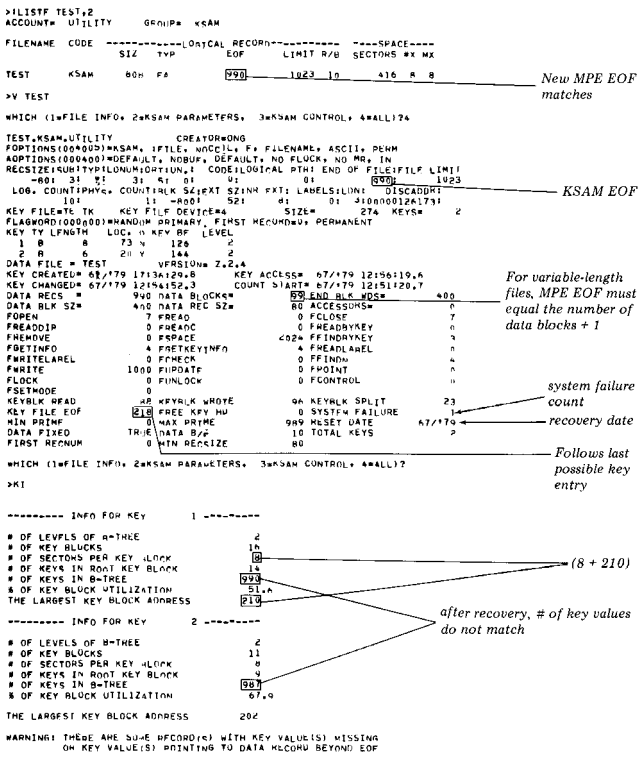
Next, check the total number of key values for each key. The first thing to notice is that they do not match each other. The number of key values for each key should always be the same, and each should equal the number of records in the data file. But, if you look at option 1 of the VERIFY display, the number of records in the data file is 990, less than the number of key

values for either key. (Note that if the file contains records marked for deletion, you can run FCOPY to determine the number of *active* records).

In response to this discrepancy, KEYINFO deletes 10 key values from each key. The values deleted are those that have no matching data record. This completes the KEYINFO functions. Now that it has deleted 10 key values from the key entries for key 2, only 987 are left (997 minus 10). This is three fewer than the number of key values in key 1 (990 = 1000 −10). For this reason, KEYINFO must issue the warning that the file needs to be reloaded:

```
--------- KEY SEQUENCE            1 ---------
# OF INVALID KEY VALUES DELETED            10<-----10 values deleted that
                                                    have no matching data record.
--------- KEY SEQUENCE            2 ---------        |
# OF INVALID KEY VALUES DELETED            10<-------|
RECOVERY ENDS

WARNING: THERE ARE SOME RECORD(S) WITH KEY VALUE(S) MISSING
         THE KSAM FILE HAS TO BF RELOADED
```

Before reloading the file, as described below, use LISTF,2 to check the current MPE end-of-file (after recovery); run VERIFY to check the current KSAM end-of-file positions; and run KEYINFO again to see the number of key values left in each key following the previous KEYINFO recovery.



The name of the user who runs KEYINFO to recover the file for the RESET DATE shown by VERIFY is stored in the key file control block, along with his account, group, and home group (refer to Figure B-5).

RELOADING A KSAM FILE

You use FCOPY to reload a KSAM file when KEYINFO cannot recover the file. You must first use the KSAMUTIL BUILD command to build the new file, and then use FCOPY to copy the file. In general, you should use the KEY=0 option of FCOPY (see section II for a complete description of the FCOPY options for KSAM files). KEY=0 copies the file in chronological sequence so that the new file will be an exact copy of the original file, except that records marked for deletion are physically deleted from the file.

For example, to reload the file TEST to a new KSAM file, NEWTEST:

```
:RUN FCOPY.PUB.SYS
>FROM=TEST;TO=(NEWTEST,NEWKEY) ;KEY=0;NEW
```

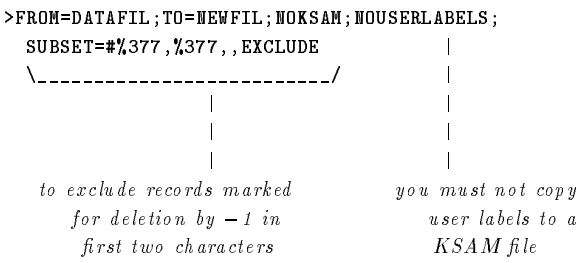
After the file is successfully reloaded, you should purge the old file TEST and rename the file NEWTEST. To do this, run KSAMUTIL and use the PURGE and RENAME commands as follows;

```
RUN KSAMUTIL.PUB.SYS
HP32208V2.4 THU, MAR 8,1979,1:05 PM            KSAMUTIL VERSION:A.3.0
>PURGE TEST
TEST,TESTKEY PURGED
>RENAME NEWTEST,TEST
>RENAME NEWKEY,TESTKEY
```

Now you can run any existing programs that referenced the old file TEST.

The only time you might not want to use the KEY= option to reload a damaged file is if the key file has been accidentally purged. In this case, and if the file has fixed-lengt,h records, you can use the NOKSAM option. This option needs only the original data file. As it copies the data file in chronological order to a new KSAM file, it creates a key file with key entries for the data records. The NOKSAM option does not, however, allow you to copy a data file with variable-length records. Refer to the instructions for copying variable-length KSAM files in Section II.

For example, to reload a KSAM data file for which you have only a data file with fixed-length records, first build the new data file, NEWFIL, then use the following FCOPY command:



This command copies only the non-deleted records; it creates a new KSAM file with only valid records and a key file that has key entries for each data record.

After a system crash in which the key file is lost, it is possible that the MPE end-of-file follows the KSAM end-of-file because it was written to disc just before the crash. If this is the case and you use the NOKSAM option you should also use a SUBSET option to copy only the records up to the KSAM end-of-file, not the undefined area between the KSAM and MPE end-of-files.

EXPAND KEY BLOCK BUFFER AREA

Depending on the length of the existing file, the reloading procedure can take a long time. One way to shorten this time is to increase the number of key block buffers in the extra data segment for the file. Since reloading is a write-only operation, the more buffers that can be allocated to key blocks, the less swapping is needed between the extra data segment and disc as new key entries are added.

In order to increase the number of key block buffers, enter the following commands:

```
:RUN FCOPY.PUB.SYS
>FROM=TEST;TO=(NEW,NEWK);SUBSET=1,0 <------------ create new file with 0 records
>:FILE F=NEW;DEV=,,20 <------------ increase number of key block buffers
>FROM=TEST;TO=*F;KEY=0 <------------ copy data in chronological order
                  |
                  |------ remember to back reference file
```

F

NATIVE LANGUAGE SUPPORT AND KSAM

OVERVIEW

The Keyed Sequential Access Method (KSAM) organizes records in a file according to the content of key fields within each record.

Native Language Support (NLS) in KSAM provides the resources to create files whose keys of type BYTE are sorted according to a native language collating sequence. All BYTE keys in the file will be sorted using the collating sequence table of the specified language. Keys, as well as data in the record, may contain 8-bit character data.

A file language attribute may be supplied when a KSAM file is created to provide a key file organized according to the collating sequence of a native language. The language attribute is provided when the file is created. All KSAM files created before NLS was introduced are considered to have NATIVE-3000 as a language attribute.

A KSAM file can bebuiltwith KSAMUTIL, or programmatically using `FOPEN`.

CREATING KSAM FILES WITH KSAMUTIL

When using KSAMUTIL, the parameter `LANG=`*langname* or `LANG=` *langnum* may be supplied on the `BUILD` command, as shown in Figure F-1. NATIVE-3000 is used as the default language attribute if no language is specified.

The language specified in the `LANG=` *parameter* must be installed on the system at the time the command is issued for KSAMUTIL to build the file. If the language is not installed, an error message is returned and the file is not built.

Danish is specified as the language in the example. The language attribute of the KSAM file can be checked by the `VERIFY` command (mode 3).

```
RUN KSAMUTIL.PUB.SYS

HP32208A.03. 13 THU,  FEB 16, 1984,  8:54 AM    KSAMUTIL VERSION:A.03.13
>BUILD TEST;REC=-;80,3,F,ASCII;KEY=B,1,4;KEYFILE =TESTK;LANG=DANISH
>VERIFY

WHICH (1=FILE INFO,  2-KSAM PARAMETERS,  3-KSAM CONTROL,  4=ALL)?4

TEST.LORO.NLS             CREATOR-SLORO
FOPTIONS(004005)=KSAM, :FILE, NOCCTL, F, FILENAME, ASCII, PERM
AOPTIONS(000400)=DEFAULT, NOBUF, DEFAULT, NO FLOCK, NO MR, IN
RECSIZE:SUB:TYP:LDNUM:DRT:UN.:  CODE:LOGICAL PTR: END OF FILE:FILE LIMIT
    -80:  9:  0:    3: 89:  2:     0:          0:           0:      1023
 LOG. COUNT:PHYS. COUNT:BLK SZ:EXT SZ:NR EXT: LABELS:LDN:   DISCADDR:
          0:          0:  -240:    43:     8:      0:  3:00000234251:

KEY FILE=TESTK    KEY FILE DEVICE=4         SIZE=       114 KEYS=      1
FLAGWORD(000020)=RANDOM PRIMARY, FIRST RECORD=0, PERMANENT
KEY TY LENGTH   LOC. D KEY BF  LEVEL
  1  B      4      1 N    168      1

DATA FILE = TEST    VERSION= A.3.13
KEY CREATED= 47/'84  9: 0: 7.6       KEY ACCESS= 47/'84    9: 0:19.2
KEY CHANGED= 47/'84  9: 0: 8.5       COUNT START= 47/'84   9: 0: 8.6
DATA RECS  =      0 DATA BLOCKS=       0 END BLK WD=            0
DATA BLK SZ=    120 DATA REC SZ=      80 ACCESSORS=             0
FOPEN             1 FREAD               0 FCLOSE                 1
FREADDIR          0 FREADC              0 FREADBYKEY             0
FREMOVE           0 FSPACE              0 FFINDBYKEY             0
FGETINFO          1 FGETKEYINFO         0 FREADLABEL             0
FWRITELABEL       0 FCHECK              0 FFINDN                 0
FWRITE            0 FUPDATE             0 FPOINT                 0
FLOCK             0 FUNLOCK             0 FCONTROL               0
FSETMODE          0 FREE KEYBLK         0 FREE RECS              0
KEYBLK READ       2 KEYBLK WRITTEN      0 KEYBLK SPLIT           0
KEY FILE EOF     10 FREE KEY HD         0 SYSTEM FAILURE         0
MIN PRIME         0 MAX PRIME           0 RESET DATE
DATA FIXED     TRUE DATA B/F            3 TOTAL KEYS             1
FIRST RECNUM      0 MIN RECSIZE         4 LANG               DANISH

WHICH (1-FILE INFO, 2-KSAM PARAMETERS,   3-KSAM CONTROL,  4-ALL)?

>E

END OF PROGRAM
:
```

Figure F-1. KSAM File Test Program

ERROR MESSAGES

KSAMUTIL error messages are listed in Table F-1.

Table F-1. KSAMUTIL Error Messages

ERROR#	MESSAGE	CAUSE	ACTION
1070	'LANG' NOT FOLLOWED BY '=' OR HAS TOO MANY PARAMETERS.	Improper syntax was used in specifying the language name.	Enter language name using correct syntax.
1071	'LANG' LANGUAGE VALUE TOO LONG OR ABSENT.	Language name too long, or missing as a parameter.	Enter correct language name.
1072	'LANG' LANGUAGE NUMBER VALUE INVALID.	The language number contains invalid characters.	Enter correct language number.
1073	'LANG' LANGUAGE NOT SUPPORTED.	Language specified is not configured on your system, or not a valid language name or number.	Ask the System Manager to configure the langauage on your system.
1074	NATIVE LANGUAGE SUPPORT IS NOT INSTALLED.	NLS is not installed on your system.	Ask the System Manager to configure the language on your system.
1075	NATIVE LANGUAGE SUPPORT LANGUAGE NOT SUPPORTED.	An NLS MPE error occurred. No language table exists for language specified.	Ask the System Manager to configure the language on your system.
1076	NATIVE LANGUAGE SUPPORT RELATED ERROR.	An NLS MPE error occurred.	Ask the System Manager to configure the language on your system; if it is already configured, contact your Hewlett-Packard support representative.

CREATING KSAM FILES PROGRAMMATICALLY

The user must provide a language ident ification number when calling `FOPEN` to build a KSAM file. The language identification number is stored in word 10 of the `KSAMPARAM` array. The `FOPEN` intrinsic checks each time a KSAM file is opened to determine whether the language used is configured on the system. For backward compatibility reasons bit 11 in the flagword (word 15) must be set to 1 if a language other than 0 (NATIVE-3000) is used, to denote that word 10 contains valid information.

If bit 11 of flagword is 0, the default language, NATIVE-3000, is used and the data in word 10 is ignored. If the language is not configured, condition code CCL is returned by `FOPEN` .

The file system error messages listed in Table F-2 have been included with NLS:

Table F-2. KSAM File System Error Messages

ERROR#	MESSAGE	CAUSE	ACTION
196	`LANGUAGE NOT SUPPORTED`	The language name or number specified for `FOPEN` is not configured on your system, or is not a valid language name or number.	Ask the System Manager to configure the language on your system.
197	`NATIVE LANGUAGE SUPPORT RELATED ERROR.`	An NLS MPE error occurred on a FOPEN call.	Contact your Hewlett-Packard support representative.

MODIFYING KSAM FILES

Every record added or updated in a KSAM file has its new keys of type BYTE inserted in the key file according to the collating sequence of the language defined for that KSAM file. That function is handled internally by a system version of the `NLCOLLATE` intrinsic when the language attribute ofthe file is different from NATIVE-3000. A new key in a file with a NATIVE-3000 language attribute will be ordered according to the result of a BYTE COMPARE between the key of the new record and the keys of the records already in the key file.

GENERIC KEYS

NLS collating sequences differ from the USASCII collating, and the differences must be considered when performing generic key searches. Refer to the *Native Language Support Reference Manual* (32414-90001) for more information.

The description of a generic key search in a KSAM file with a native language attribute is presented from an application point of view

Keys matching a certain generic key may not be in consecutive order in the key file because the keys are sorted according to a native language collating sequence. The key sequence in Figure F-3 illustrates this with a French KSAM file; *keylength* is 4, the generic key length is 2. The partial key "aa" appears in nonconsecutive keys (with a result of 0 in the last column of the figure). Records containing partial keys (such as "AA" or "Aa"0 are intermixed according to the French collating sequence. These keys have a result of 1 listed.

If a generic key search is performed in a KSAM file with a language attribute other than NATIVE-3000, the application program must determine whether the retrieved record matches the generic key and, even if it does not, whether subsequence records might still match it.

The codes returned by `NLKEYCOMPARE` are shown in Figure F-2.

Refer to the *Native Language Support Reference Manual* (32414-90001) for a complete discussion of the `NLKEYCOMPARE` intrinsic.

Figure F-2. Results Returned By The `NLKEYCOMPARE` Intrinsic

RESULT	MEANING
0	The retrieved key matches the generic key exactly.
1	The retrieved key does not match the generic key. Uppercase/lowercase priority or accent priority is different.
2	The retrieved key value is less than the generic key. It precedes the designated key in the collating sequence.
3	The retrieved key is greater than the generic key.

The generic key search sequence is:

1. After `FFINDBYKEY` has been called with >= as relational operator(relop), the logical record pointer points to the data records indicated by the arrow labeled "Case 2".
2. The subsequent `FREAD` call will retrieve the data record. When the partial key "AA" is compared to the generic key "aa" they are found to be different.

   This comparison is done by calling the intrinsic `NLKEYCOMPARE` using the generic key and the key found in the record. The result returned by `NLKEYCOMPARE` tells the application whether the `FREAD` delivered a record:

   a. Before the desired range (result 2).

   b. In the desired range with an uppercase/lowercase or accent priority difference (result 1).

   c. With an exact match (result 0).

   d. After the desired range (result 3).
3. To get all records whose key match the generic key exactly, the `FREAD` calls and subsequent `NLKEYCOMPARE` call should continue until a result of 3 is returned.

When performing a generic key search in a KSAM file with a native language attribute other than NATIVE-3000 use the `NLKEYCOMPARE` intrinsic to compare partial keys and generic keys.

Refer to the *Native Language Support Reference Manual* (32414-90001) for examples of generic key searches in KSAM files with native language attributes.

Key length: 4

Language: FRENCH (only USASCII characters are used in the example).

Desired records are all records whose record key starts with "**aa**" (generic key = "**aa**", length = 2) .

Pointer Position	Key Value	NLKEYCOMPARE Result ("aa" Compared to Key)
Case 1—->	**A**	2
	a	2
Case 2—->	**AA**	1
	Aa	1
	aA	1
	aa	0
	AAA	1
	aaa	0
	AAAA	1
	AAAa	1
	AAaa	1
	AaAa	1
	AaaA	1
	Aaaa	1
	aAAA	1
	aAAa	1
	aAaA	1
	aaAA	0
	aaaA	0
	aaaa	0
Case 3—->	**Baaa**	3
	baaa	3

Figure F-3. Generic Key Searches

Case:

1. `FREAD` starting at the beginning of the file.

2. `FFINDBYKEY` with relational operator = or >= and subsequent `FREAD` calls.

3. `FFINDBYKEY` with relational operator > and subsequent `FREAD` calls.

Key Value: Key values in ascending sequence.

Figure F-4. Generic Key Searches

USING FCOPY WITH NLS KSAM FILES

COPYING FROM A KSAM FILE TO ANOTHER KSAM FILE

If the KSAM file already exists (built via KSAMUTIL or programmatically) the keys of type BYTE are put into the new file according to the collating sequence belonging to the language of the "TO" file. If the file does not exist, a new file is built with the same language attribute as the "FROM" file.

CHANGING THE LANGUAGE ATTRIBUTE OF A KSAM FILE

FCOPY cannot be used to change the language attribute of an existing file. KSAMUTIL must be used to build a new KSAM file with the new language attribute. Then the data can be copied to this file using FCOPY. Keys of type BYTE in the destination key file will be ordered according to the collating sequence of the new language.

MOVING NLS KSAM FILES TO PRE-NLS MPE

Restoring a KSAM file with a native language attribute other than NATIVE-3000 to a system without NLS installed can result in an incorrect key sequence in the key file for tvpe BYTE keys. Systems without NLS installed do not recognize any collating sequence except NATIVE-3000.

If a file with a native language attribute otherthan NATIVE-3000 is restored, the first `FOPEN` on the file will return the same error condition code as if a system failure occurred while the file was opened. KSAMUTIL should be used to build a new KSAM file, The file with the native language attribute is recovered, and FCOPY is used to copy the recovered file into the new KSAM file. See Figure F-5 for an example of this recovery procedure.

```
:RUN KSAMUTIL.PUB.SYS

HP32208A.03.10  SAT,  SAT,  MAY 26,1984,  12:33 PM  KSAMUTIL VERSION:A.03.10
>BUILD NEWDATA;REC=-80,3,F,ASCII;KEY=B,1,4,:KEYFILE =NEWKEY
>KEYINFO OLDDATA;RECOVER

>EXIT

:FCOPY FROM-OLDDATA;TO-NEWDATA;KEY=0

:RUN KSAMUTIL.PUB.SYS

HP32208A.03.10 SAT, SAT,MAY 26,1984, 12:33 PM KSAMUTIL VERSION:A.03.10
>PURGE OLDDATA
>RENAME NEWDATA,OLDDATA
>RENAME NEWKEY,DLDKEY
>EXIT
```

Figure F-5. KSAM Recovery Procedure