FORTRAN for NOS/VE
Keyed-File and Sort/Merge Interfaces

Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

Publication Number 60485917

Manual History

Revision	System Version	AAM Product Level	Sort/Merge Product Level	PSR Level	Date
A	1.4.1	1.7	1.3	716	December 1988
B	1.5.1	1.8	1.3	739	December 1989

This revision documents:

- A new file-spanning parcel feature, which allows you to group a series of update operations that apply to more than one keyed file.
- Miscellaneous technical and editorial changes.

©1988, 1989 by Control Data Corporation
All rights reserved.
Printed in the United States of America.

Contents

About This Manual 5

- Audience 5
- Organization 6
- Conventions 7
- Ordering Printed Manuals 8
- Submitting Comments 8
- In Case You Need Assistance 9

Keyed-File Interface Concepts 1-1

- General Keyed-File Concepts 1-2
- FORTRAN Keyed-File Interface Concepts 1-16

Alternate Keys 2-1

- What Are Alternate Keys? 2-1
- How to Use Alternate Keys in FORTRAN Programs 2-15

Sharing Keyed Files 3-1

- When Does Sharing Keyed Files Require Locks? 3-2
- What Are Access and Share Modes? 3-3
- What Are Locks? 3-9

Parcels 4-1

- File-Spanning Parcels 4-1
- How to Use Parcels in FORTRAN Programs 4-2
- File-Level Parcels 4-10
- Parcel Program Example 4-11

Result Sets 5-1

- What Are Result Sets? 5-1
- How to Use Result Sets in FORTRAN Programs 5-2

Keyed-File Interface Calls 6-1

- How to Use Keyed-File Interface Calls in FORTRAN Programs 6-2
- Keyed-File Interface Calls: Quick Reference 6-5

File Information Tables 7-1

- How to Use FIT Keywords in FORTRAN Programs 7-3
- FIT Keywords and Values: Quick Reference 7-4

Sort/Merge Interface 8-1

- What Sort/Merge Does 8-1
- Sort Keys 8-2
- Defining Sort Keys 8-3
- Specifying the Record Length 8-10
- Performance Considerations 8-15
- Sort/Merge Procedure Calls 8-16.1
- Owncode Procedures 8-52
- Owncode 1: Processing Input Records 8-56
- Owncode 2: Processing Input Files 8-58
- Owncode 3: Processing Output Records 8-59
- Owncode 4: Processing the Output File 8-61
- Owncode 5: Processing Records With Equal Keys 8-62
- Using FORTRAN Procedure Calls 8-64
- Creating an Object Library 8-70
- Summing Records 8-72
- Defining Your Own Collating Sequence 8-73

Glossary A-1

Related Manuals B-1

- Ordering Printed Manuals B-1
- Accessing Online Manuals B-1

ASCII Character Set and Collating Weight Tables C-1

Creating a Collation Table D-1

- Predefined Collation Tables D-1
- Creating Your Own Collation Table D-2
- Collation Table Example D-3

Creating a Collation Weight Table.......................... D-10

Differences Between NOS/VE FORTRAN and FORTRAN 5..... E-1

CYBER Record Manager (AAM) Subprograms.................... E-1

Default Collating Sequence........ E-4

Other Collating Sequence Differences...................... E-4

Sort/Merge E-4

Index Index-1

Figures

1-1. Minimal Indexed-Sequential Structure.......................... 1-4

1-2. Data Block Split 1-6

1-3. Index Block Split (Part a)....... 1-8

1-4. Index Block Split (Part b)....... 1-9

8-1. Internal Data Representation 8-4

8-2. When Owncode Procedures are Called........................... 8-53

8-3. Output From the FORTRAN Program.......................... 8-67

8-4. Output From Program INMEMORYSORT................. 8-69

D-1. Creation Program D-3

D-2. Creation Program Output....... D-8

D-3. Collation Weight Table D-11

Tables

2-1. Data Records With Duplicate Alternate-Key Values............... 2-1

2-2. Data Record With Several Alternate Keys..................... 2-1

2-3. Ordered by Primary Key 2-4

2-4. Ordered by First-In-First-Out ... 2-4

3-1. When the Lock Request is From the Same Instance of Open.. 3-17

3-2. When the Lock Request is From Another Instance of Open.... 3-17

3-3. When the Lock Request is From the Same Instance of Open.. 3-18

3-4. When the Lock Request is From Another Instance of Open.... 3-18

6-1. FIT Keywords That Can Be Fetched on a Closed File.......... 6-20

8-1. Maximum Key Field Sizes 8-3

8-2. Numeric Data Formats 8-6

8-3. Sign Overpunch Representation . 8-9

8-4. Result Array Format 8-34

B-1. Related Manuals B-2

C-1. ASCII Character Set and Collating Sequence................. C-2

C-2. OSV$ASCII6_FOLDED Collating Sequence................. C-6

C-3. OSV$ASCII6_STRICT Collating Sequence................. C-8

C-4. OSV$COBOL6_FOLDED Collating Sequence................ C-10

C-5. OSV$COBOL6_STRICT Collating Sequence................ C-12

C-6. OSV$DISPLAY63_FOLDED Collating Sequence................ C-14

C-7. OSV$DISPLAY63_STRICT Collating Sequence................ C-16

C-8. OSV$DISPLAY64_FOLDED Collating Sequence................ C-18

C-9. OSV$DISPLAY64_STRICT Collating Sequence................ C-20

C-10. OSV$EBCDIC Collating Sequence........................ C-22

C-11. OSV$EBCDIC6_FOLDED Collating Sequence................ C-29

C-12. OSV$EBCDIC6_STRICT Collating Sequence................ C-31

About This Manual

This manual describes the CONTROL DATA® FORTRAN procedures that NOS/VE provides to use the keyed-file and Sort/Merge interfaces.

This manual is a usage manual. It is functionally organized to describe how to use the keyed-file and Sort/Merge Interfaces within a FORTRAN program.

To use the keyed-file and Sort/Merge interfaces from the NOS/VE command level, see the NOS/VE Advanced File Management manual.

Audience

You should be familiar with the FORTRAN language as described in the FORTRAN for NOS/VE Version 1 or Version 2 Language Definition manuals. In addition, you should know how to create and run jobs under the NOS/VE operating system. These concepts are described in the Introduction to NOS/VE manual and, in more detail, in the NOS/VE System Usage manual.

You should be familiar with keyed files and sort/merge procedures. Although this manual discusses general concepts of keyed files and sort/merge procedures, it is not a tutorial.

Organization

The FORTRAN manual set consists of the following manuals:

FORTRAN for NOS/VE Tutorial

This manual is intended for the programmer who has no previous FORTRAN experience. It presents a tutorial introduction to the FORTRAN language, beginning with the basic elements of the language and proceeding through more complex features.

FORTRAN for NOS/VE Topics for Programmers

This manual is intended for experienced FORTRAN programmers who are new to NOS/VE. It presents introductory topics intended to help FORTRAN programmers use the NOS/VE operating system and NOS/VE FORTRAN effectively. Topics covered include System Command Language, debugging, input/output, optimization, virtual memory, and object libraries.

FORTRAN for NOS/VE Summary

This manual presents a concise pocket-size summary of the FORTRAN language. It presents a complete list of FORTRAN statements in alphabetical order and shows the parameters for each statement. It does not include detailed parameter descriptions.

FORTRAN Version 1 for NOS/VE Quick Reference (Online)

This manual provides an online quick reference for the FORTRAN Version 1 commands, statements, functions, and subprograms. Parameter descriptions and examples are included. This manual also explains all compilation diagnostics. To access this manual, enter

```
/help m=fortran
```

FORTRAN Version 1 for NOS/VE Language Definition Usage

This manual presents detailed descriptions and definitions of all the statements and features of the NOS/VE FORTRAN Version 1 language. Examples of statements and programs are included.

FORTRAN Version 2 for NOS/VE Quick Reference (Online)

This manual provides an online quick reference for the FORTRAN Version 2 commands, statements, functions, and subprograms. Parameter descriptions and examples are included. This manual also explains all compilation diagnostics. To access this manual, enter

```
/help m=vfortran
```

FORTRAN Version 2 for NOS/VE Language Definition Usage

This manual presents detailed descriptions and definitions of all the statements and features of the NOS/VE FORTRAN Version 2 language. FORTRAN Version 2 can use the vectorization capabilities of the CYBER 990 or 995 class mainframes to improve a program's execution time. Examples of statements and programs are included.

FORTRAN for NOS/VE LIB99 Usage

This manual presents a library of subroutines and functions that can be called from both FORTRAN Version 1 and FORTRAN Version 2. With LIB99 routines, you can perform vector arithmetic and matrix algebra, solve linear systems of equations, compute Fast Fourier Transforms, compute eigenvalues and eigenvectors, and sort lists.

Conventions

All numbers used in this manual are decimal unless otherwise indicated. Other number systems are indicated by a notation after the number. For example, 177 octal, FA34 hex.

Certain notations are used throughout the manual with consistent meaning. The notations are:

Integer	Unless otherwise specified, the term integer indicates an 8-byte integer.
UPPERCASE	In language syntax, uppercase indicates a statement keyword or character that is to be written as shown. Although lowercase letters are interpreted the same as uppercase characters when used in FORTRAN keywords and symbols, uppercase is used for consistency. In occasional examples, keywords and symbols are shown in lowercase for illustrative purposes.
lowercase	In language syntax, lowercase indicates a name, number, symbol, or entity that you must supply.
`computer font`	Indicates text of examples.
Boldface	In language syntax, boldface type indicates a required keyword, parameter, or symbol.
Italics	In language syntax, optional keywords, parameters, and symbols are shown in italics.
...	In language syntax, a horizontal ellipsis indicates that the preceding optional item can be repeated as necessary.
⋮	In program examples, a vertical ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.
Δ	Space character. This symbol is used wherever there might otherwise be doubt as to how many spaces are intended.

Vertical bars in the margin indicate changes or additions to the text from the previous revision. An example of a change bar is shown in the margin next to this paragraph.

Ordering Printed Manuals

Control Data manuals are available through your local Control Data sales offices. Sites within the U.S. can also order manuals directly from Control Data Literature Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

When ordering a manual, please specify the complete title, publication number, revision level, and whether you want the complete manual or the latest revision packet.

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation
Technical Publications
5101 Patrick Henry Drive
Santa Clara, California 95054-1111

Please indicate whether you would like a written reply.

Be sure to include the following information with your comment:

FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces Usage manual
Publication number 60485917
Current revision letter (from the Manual History)

Also, if you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit comments about this manual. When it prompts you for a product identifier for your report, please specify AA8 when commenting on the keyed-file interface and SM8 when commenting on the Sort/Merge interface.

In Case You Need Assistance

Control Data's CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or you can call (612) 292-2101. If you are a Control Data employee, call CONTROLNET® 243-2100 or (612) 292-2100.

Keyed-File Interface Concepts 1

General Keyed-File Concepts 1-2
What Is a Keyed File? 1-2
How Are Keyed Files Organized? 1-2
Indexed-Sequential File Organization 1-3
Indexed-Sequential File Structure 1-3
Data-Block Split 1-6
Index Levels 1-8
Indexed-Sequential Primary Keys 1-10
Direct Access File Organization 1-11
Direct Access File Structure 1-12
Hashing Procedure 1-14
Direct Access Primary Keys 1-15

FORTRAN Keyed-File Interface Concepts 1-16
How to Use Keyed Files in FORTRAN Programs 1-16
Using File Information Tables 1-16
Creating a Keyed File 1-17
Keyed-File Attributes 1-18
Using an Existing Keyed File 1-19
Processing Errors 1-20

Keyed-File Interface Concepts 1

The keyed-file interface is a group of subprogram calls that use the NOS/VE keyed-file interface to operate on keyed files. The following section, General Keyed-File Concepts, describes keyed-file structure. This information applies when using keyed files within or outside of programs. The next section, FORTRAN Keyed-File Concepts, describes concepts unique to the FORTRAN keyed-file interface.

General Keyed-File Concepts

This section describes concepts of keyed files that are not specific to FORTRAN or NOS/VE. The discussion is limited to indexed-sequential and direct-access keyed files. For information on other types of files, such as sequential files, see the FORTRAN for NOS/VE Version 1 or Version 2 Language Definition manuals.

What Is a Keyed File?

A keyed file is a file whose organization allows you to access records by their key values.

Keyed files are similar to sequential files and byte-addressable files in that the data in the files is contained in records.

A record is a collection of data that is read and written as a unit. The record could contain several fields of data, some of which have a fixed length while others vary in length. Thus, the records as a whole could have a fixed length or be variable in length.

For example, a record could contain three data items of different types: an integer, a floating point number, and a string of characters. To write a record, a program writes all three data items together as a record; when the record is later read, all three data items are delivered to the program.

The records in a sequential or byte-addressable file are stored as a simple sequence. The records in a keyed file are stored within a file structure.

How Are Keyed Files Organized?

NOS/VE keyed files are organized as indexed sequential or direct access. Therefore, a file is a keyed file if its file_organization attribute is either indexed_sequential or direct_access.

You can display a file's organization with the NOS/VE command DISPLAY_FILE_ATTRIBUTES. This example displays the organization of a file named INDEXSEQ:

```
/display_file_attributes file=indexseq display_options=file_organization

File_Organization                    : indexed_sequential
```

A keyed-file organization allows you to read any record in the file directly by specifying its key value. The key value for a record is determined when the record is written to the file.

To allow you to access each record by a key value, the file organization must relate each key value to the location of the record in the file. The keyed-file interface performs all processing required to relate a key value to a record location; beyond choosing the file organization, the user does not specify how this is done. The method of relating a key value to a record location differs for each keyed-file organization as described in the following sections.

Indexed-Sequential File Organization

An indexed-sequential file stores records in sequential order, sorted by primary-key values. If you read an indexed-sequential file sequentially, the records are returned in order. This method is more efficient than direct-access files for sequential reads when you want the records returned in primary-key value order.

An indexed-sequential file always has a primary key. It can also have one or more alternate keys as described in the Alternate Keys section of this chapter.

Each primary-key value is unique within the file; there can be no duplicate primary-key values in a file.

The indexed-sequential file organization is used only when you can assign a unique value to each record stored in the file. This unique value is usually a field of data within the record (an embedded key), although it can be a value assigned to the record and not included in the record data (a nonembedded key).

For example, the primary key for an employee file could be the employee's name. However, because two employees could have the same name, it is better to assign a unique identification number to each employee and use that number as the primary key for the file.

You should use the indexed-sequential file organization if you need to read records both sequentially and randomly. For example, the records in an employee file could be read sequentially to produce a listing of all employees or read randomly to update individual records.

When an indexed-sequential file is read sequentially, its records are accessed in ascending order by key value. The order is kept even when new records are added to the file. For example, if an employee file is read sequentially using its primary key (the employee identification number) the records are read in ascending order by their identification number.

Indexed-Sequential File Structure

This subsection gives a general description of the indexed-sequential structure. You can use indexed-sequential files without knowing their structure. However, if you understand the indexed-sequential structure and how it grows, you can create more efficient indexed-sequential files by specifying appropriate values for structural parameters.

The internal structure of an indexed-sequential file is designed to provide both random and sequential access to the data records in the file. File space is divided into blocks of equal size.

A block contains a block header and one of the following:

- Internal tables
- Data records (a data block)
- Index records (an index block)

Each index record points to a data block. The index record contains the location of the data block and the range of key values of the data records stored in that block.

You can display the contents of all components of an indexed-sequential file, the internal tables and index blocks as well as the data blocks, using the DISPLAY_KEYED_FILE command described in the NOS/VE Advanced File Management Usage manual.

As you might expect, the actual internal index mechanism is complex. The simplified examples in this part, however, provide the level of detail you need in order to use indexed-sequential files.

To see how an index works, let's look at a very small file that contains one index block and two data blocks. As shown in figure 1-1, the index block contains two index records. Each index record points to a data block in the file.

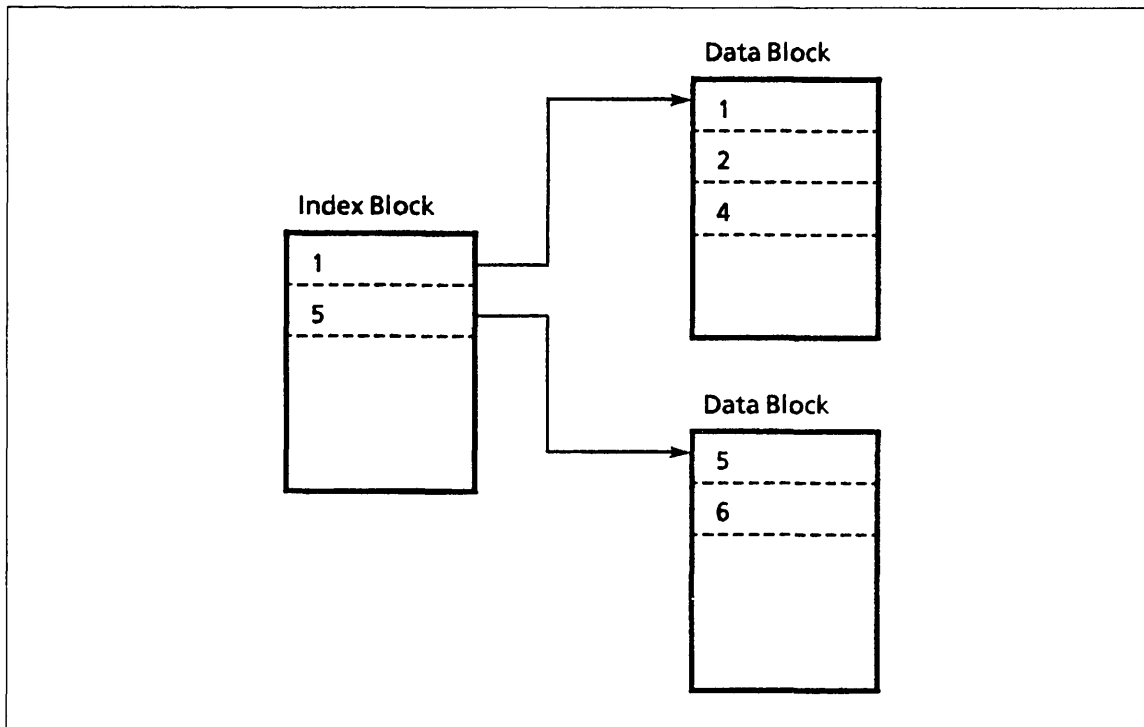


Figure 1-1. Minimal Indexed-Sequential Structure

Let's suppose you request to read the record with key value 6. To find the record, these steps are performed:

1. The index records are searched to find the index record whose range of key values includes the key value 6.
2. After the correct index record (the second one) is found, the search for the record continues with the data block pointed to by the second index record.
3. The second data block is searched for the record with key value 6. When the record is found, its data is returned to the requester.

Next, suppose you request to sequentially read all records in the file. These steps are performed:

1. The first index record is read to find the first data block.
2. The records from the first data block are read in order.
3. The second index record is read to find the second data block.
4. The records from the second data block are read in order.
5. The sequential read ends because there are no more index records and therefore no more data blocks to read.

This process reads the records in key-value order because both the index records and the data records are kept in key-value order.

Data-Block Split

Usually, a block has some empty space, called padding, that was left empty so that additional records could later be written to the block. Suppose, as shown in figure 1-2, a new record is to be written, and its key value is within the range of key values of the records in a full data block. For the file structure to be maintained, the data block must be split.

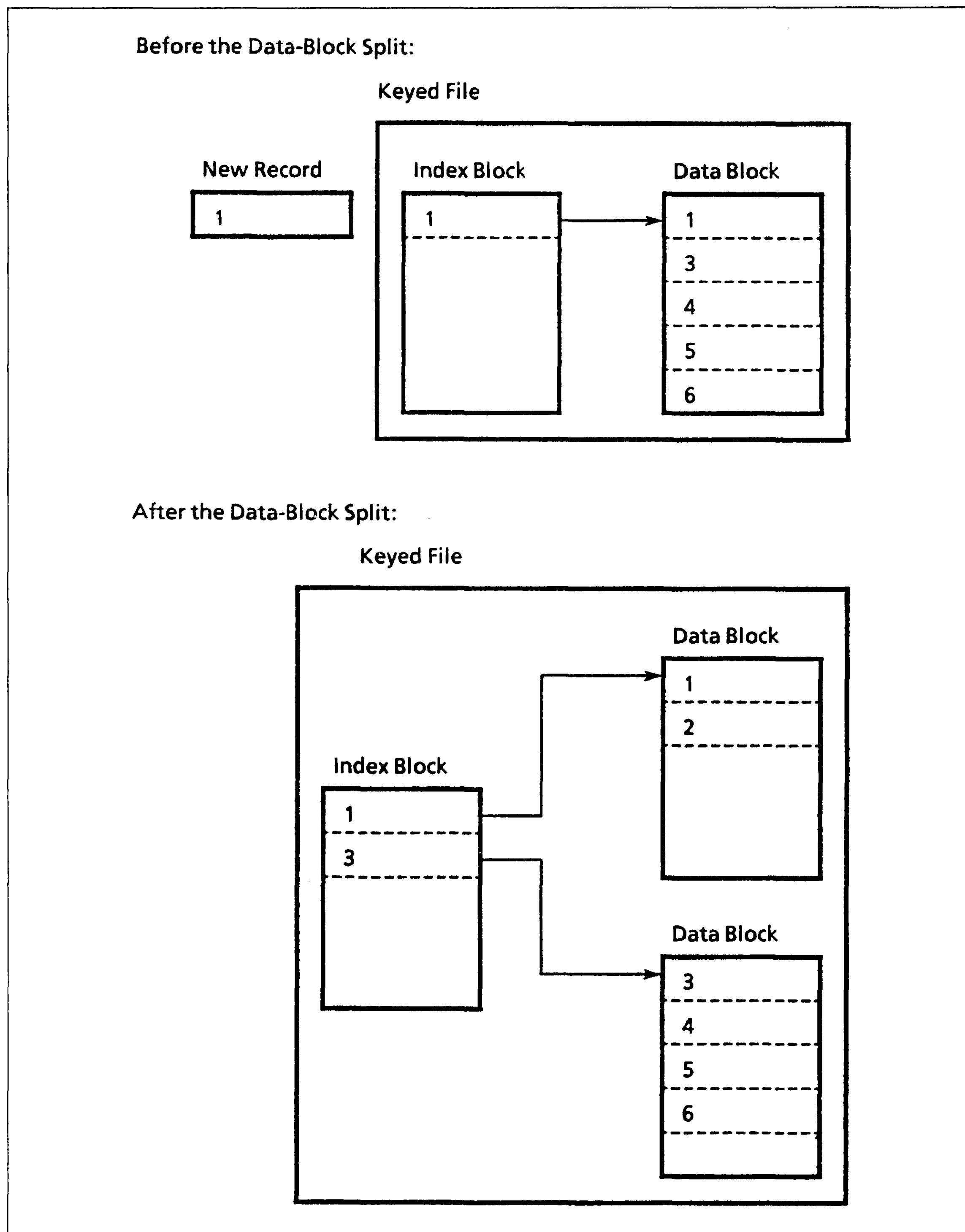


Figure 1-2. Data Block Split

When a data-block split occurs, records in the data block whose key values are less than the key value of the new record remain in the existing block. All records in the existing block that come after the new record are moved to the newly created block.

The new record is put into either the new block or the existing block, depending on the relative amount of empty space in the blocks and the size of the new record. If the new record does not fit in either block, a second new block is created and the new record is put into that block.

Index Levels

As with data blocks, index blocks may be initially created with some empty space known as index-block padding. However, for each new data block created due to a data-block split, another index record must be created. With the addition of many data records, the initial index block becomes full. When the index block is full, the next data-block split causes an index-block split.

As shown in figure 1-3, when the initial index block splits, it causes the creation of another index level.

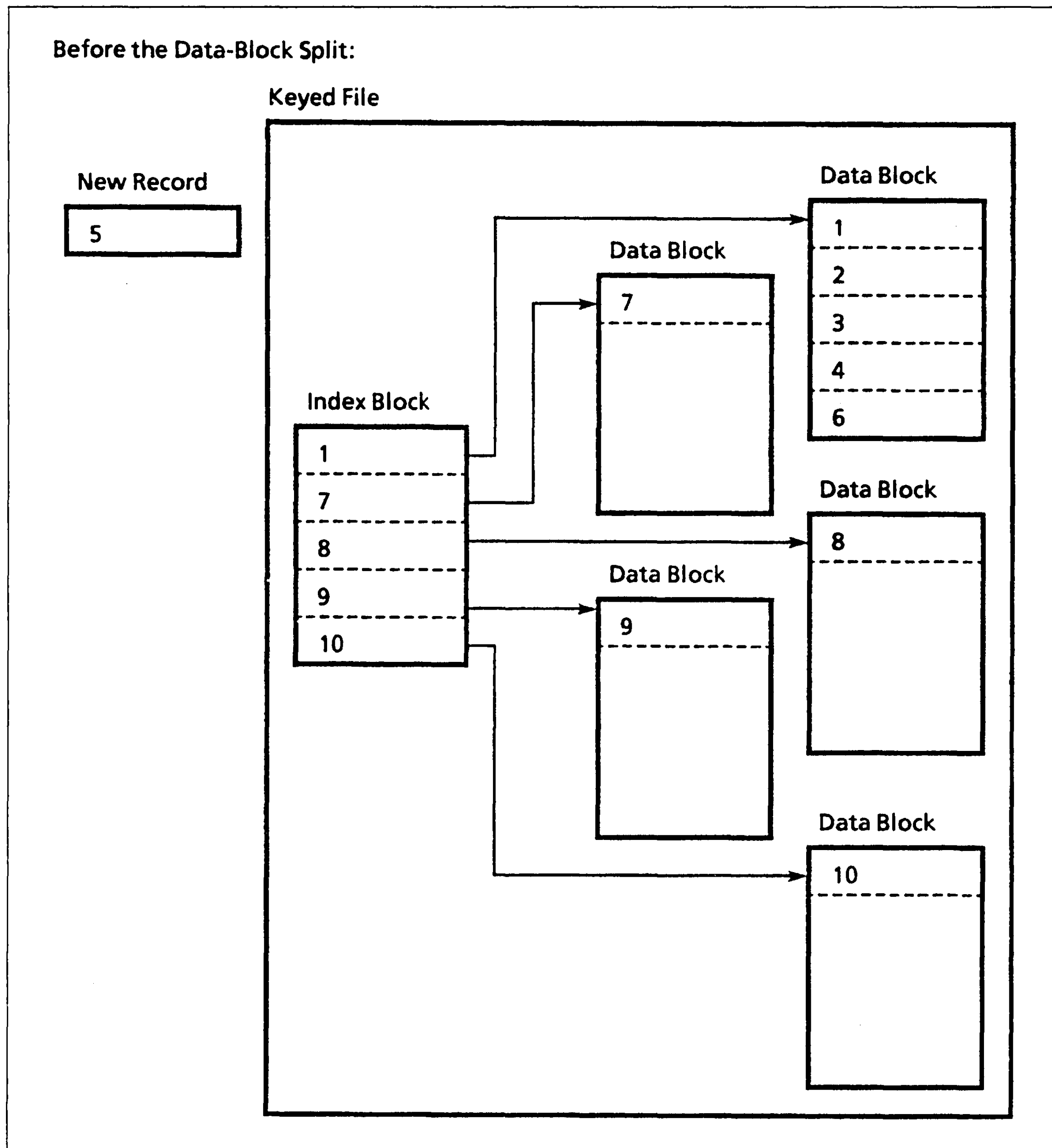


Figure 1-3. Index Block Split (Part a)

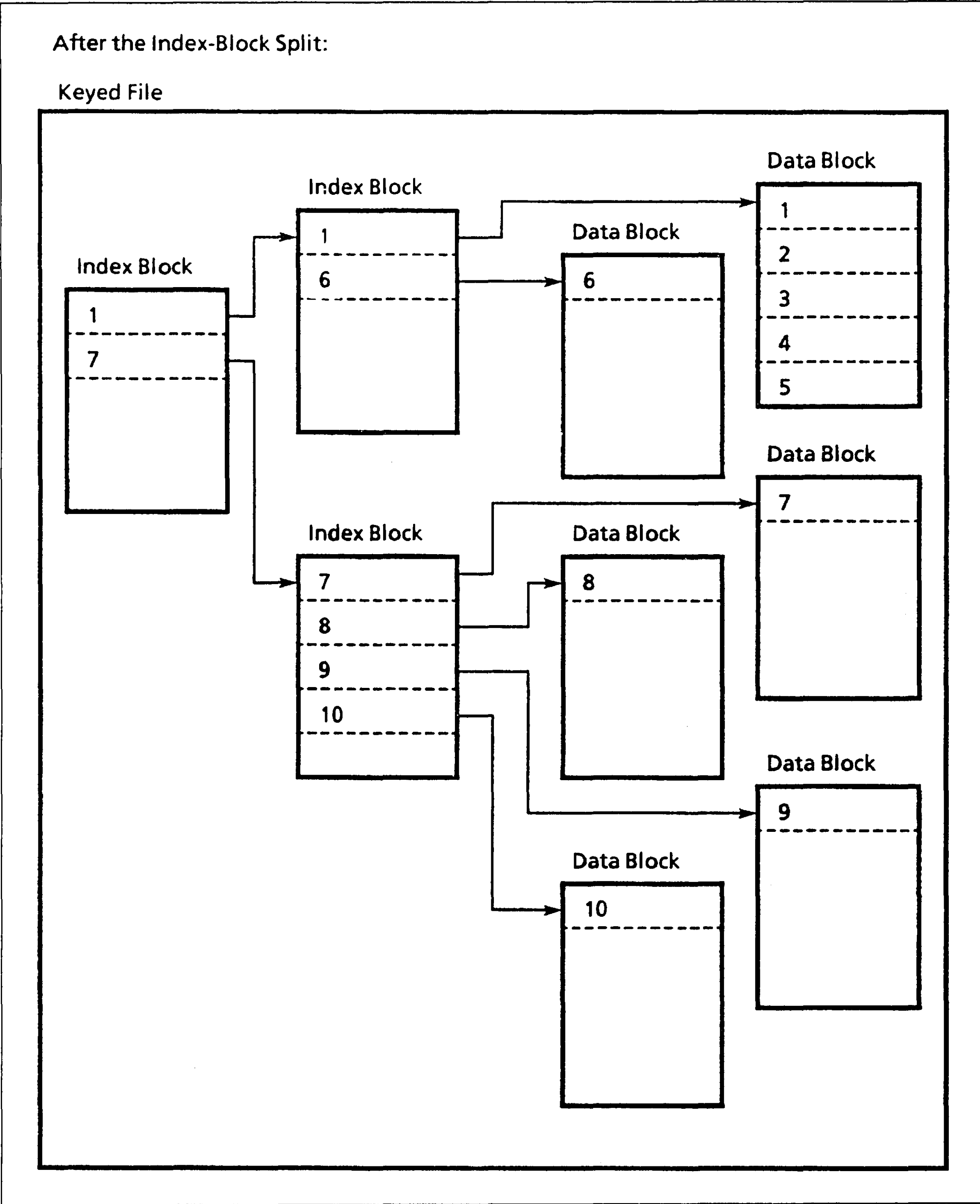


Figure 1-4. Index Block Split (Part b)

The index levels are numbered as index level 0, index level 1, and so forth. Index level 0 always has only one index block; it is always the starting point for an index search.

The index block at an upper level contains an index record for each index block at the next lower level. For example, the index block at level 0 contains an index record for each index block at level 1.

A search for a data record requires an index-block search at each index level. For example, the level-0 search finds the index record that points to the appropriate level-1 index block. If the file has only two index levels, the level 1 search finds the index record that points to the appropriate data block.

As you can see, the addition of another index level increases the time required to find an individual data record.

Index levels can be added up to the index-level limit of 15 levels. This sets a limit on the number of records in the file.

The index-level limit is reached when addition of another record to the file would require creation of another index level, but 15 index levels already exist in the file. When this happens, the index-level-overflow flag is set and no more records can be added to the file.

Indexed-Sequential Primary Keys

The primary key for an indexed-sequential file is defined when the file is created. The primary-key value must be unique for each record in the file.

A primary-key definition requires you to specify these attributes:

- Embedded or nonembedded key (the default is embedded)
- Key position (if the key is embedded)
- Key length
- Key type (the default is uncollated)
- Collate-table name (if the key type is collated)

A key is embedded if the key value is part of the data in the record. An embedded key value is returned as part of the record data when the record is read; a nonembedded key value is not.

You must specify the key position in the record if the key is embedded. The first byte position in a record is byte 0. If the key is nonembedded, you do not specify a key position.

You must specify the key length whether the key is embedded or nonembedded. It indicates the number of bytes in the key.

The key type describes the data in the key. These are the possible key types:

Integer key

The key value is a signed integer; it is sorted in numerical order.

Uncollated key

The key value is a string of characters; it is sorted byte-by-byte according to the ASCII collating sequence.

Collated key

The key value is a string of characters; it is sorted byte-by-byte according to a collating sequence that you specify.

If the key is a collated key, you must specify the collating sequence to be used to sort the key values. The collating sequence is specified by its name. NOS/VE provides several predefined collating sequences (listed in appendix C, ASCII Character Set and Collating Weight Tables). You can also create your own collating sequence as described in appendix D, Creating a Collation Table.

Direct Access File Organization

The direct-access file organization is like the indexed-sequential file organization in its use of a primary key. You define the primary key for the file when you create the file. It can be a field embedded in the record or a nonembedded value. Each primary-key value in the file must be unique.

Like an indexed-sequential file, a direct-access file can have alternate keys. An alternate key for a direct-access file is the same as an alternate key for an indexed-sequential file. Alternate keys are described later in this chapter.

Like indexed-sequential file records, you must specify the primary-key value when writing or deleting a direct-access file record. Similarly, you must specify either a primary-key value or an alternate-key value to read a direct access file record.

Direct access and indexed-sequential files differ in the ordering of records in the file:

- When records are read sequentially from an indexed-sequential file, the records are returned in order, sorted by primary-key value.
- When records are read sequentially from a direct-access file, the records are returned unordered.

In general, random record access is faster for the direct-access file organization than the indexed-sequential file organization. This is because the direct-access file organization determines the location of a record directly from its primary-key value. In indexed-sequential files, a record can be found only after a search at each index level.

Direct Access File Structure

A direct-access file does not store records in primary-key value order like an indexed-sequential file. It uses an algorithm, called a hashing procedure, to determine where to write the records in a file. However, direct-access files are more efficient than indexed-sequential files at retrieving records in random order.

The direct-access file structure is designed to locate each record directly by its primary-key value. The primary-key value directly specifies the file block containing the record.

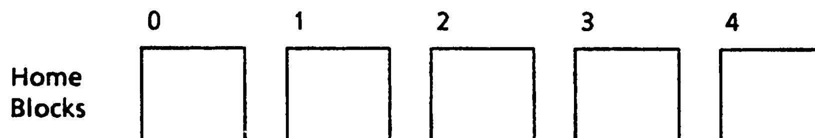
File space in a direct-access file is divided into equal-size blocks. Initially, all blocks in the file are home blocks. When a home block fills with records, records are written to overflow blocks.

When a record is written to a direct-access file, its primary-key value is hashed to produce the number of the home block in which the record is written. If the home block does not contain enough empty space for the new record, the record is written to an overflow block.

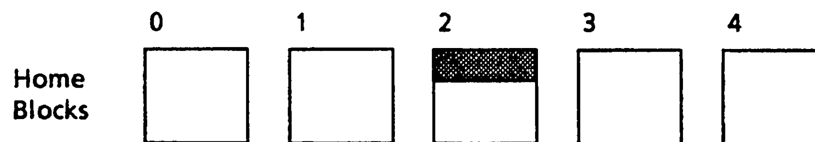
Assuming the hashing procedure produces a uniform distribution of numbers from the primary-key values in the file, the records are uniformly distributed among the home blocks of the file. Thus, each record can be found by a single search of its home block without additional searches of overflow blocks.

You specify the initial number of home blocks when you create the file. By default, a system hashing procedure is used to distribute the records among the home blocks, although you can provide another hashing procedure for the file if you like.

As an illustration of a small direct-access file, suppose you define a direct access file as having five home blocks.

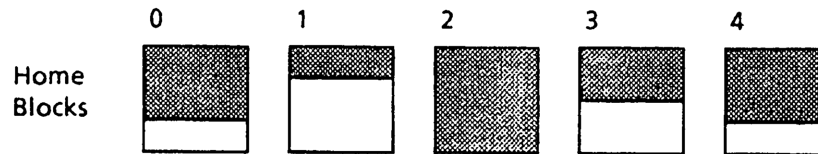


The first record written to the file has primary-key value XYZ. Assume that hashing of this primary-key value produces the block number 2. The record is then written in home block 2.

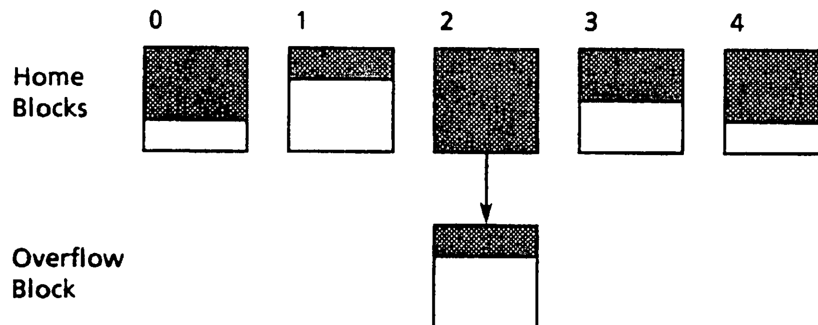


Assume you want to read the record with primary-key value XYZ. The value XYZ is hashed and, as before, produces the block number 2. The keyed-file interface searches for the record with primary-key value XYZ in home block 2. (The records in a block are ordered by primary-key value so each record can be found quickly.)

Suppose that many records have been written to the file and home block 2 has been filled.



At this point, a record is to be written with primary-key value ABC. Hashing of the value ABC produces block number 2, but there is insufficient space for the record in home block 2 so it is written in an overflow block.



Later, to read the record with primary-key value ABC, the primary-key value is hashed to produce block number 2. Home block 2 is searched for primary-key value ABC. When it is not found in the home block, the search continues in the overflow block until the record is found.

For Better Performance

An ideal direct-access file structure has these characteristics:

- Sufficient home blocks are allocated and records are uniformly distributed among the home blocks to avoid overflow.
- Each block contains a limited number of records to minimize the search time in each block.
- The number of home blocks is not so large that the file contains excessive unused space.
- The number of home blocks is prime to provide for a more even distribution of records.

These characteristics are determined by the file attributes you values specify when the file is created.

One other characteristic to be considered when selecting the number of home blocks is the loading factor. The loading factor is the percentage of block space used. To allow for less-than-uniform distribution of records in the home blocks, the loading factor should be no greater than 90%.

You can use the following equations to determine the minimum home block count for a given loading factor if the number of bytes of data in the file and the block size are known.

If the file has fixed-length records, reduce the block size by 39 bytes, as follows:

$$home_block_count = \frac{record_count \times fixed_record_length}{loading_factor \times (block_size - 39)}$$

If the file has variable-length records, reduce the block size by 36 bytes and use the average record length plus 3 as the record length, as follows:

$$home_block_count = \frac{record_count \times (average_record_length + 3)}{loading_factor \times (block_size - 36)}$$

To illustrate, suppose the direct-access file is to contain 10,000 80-byte records (800,000 bytes of record data). Using a block size of 4096 bytes and a loading factor of 90%, the equation appears as follows:

$$home_block_count = \frac{10000 \times 80}{.90 \times (4096 - 39)}$$

The equation gives 220 blocks as the minimum home block count for the file. However, it is recommended that the home block count be a prime number so 223 would be a better home block count for the file in this example.

Hashing Procedure

The system provides a default hashing procedure named AMP$SYSTEM_HASHING_PROCEDURE. However, you may specify your own hashing procedure that produces a uniform distribution of numbers from the primary-key values in your file.

The system executes the hashing procedure each time a record is requested by key value from the direct-access file. The hashing procedure is not stored with the file so the system must be able to load the procedure each time the direct-access file is opened.

For Better Performance

Although any ring attributes value is valid for the object library containing the hashing procedure, you should store the hashing procedure in a ring 4 object library. This improves performance because otherwise the hashing procedure is loaded by an asynchronous task. (Ring 4 object libraries are usually maintained by site personnel.)

A hashing procedure receives a primary-key value as its input and produces an integer as its output. It must always produce the same output from a given input.

A hashing procedure must be written in the CYBIL language. For information on how to write a hashing procedure, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.

The system divides the value it receives from the hashing procedure by the number of home blocks and uses the remainder as the home block number. For example, if the number of blocks is 97, it divides the hashed value by 97 and uses the remainder (an integer from 0 through 96) as the home block number. A more uniform distribution of records can be expected if the number of home blocks is a prime number.

Direct Access Primary Keys

In general, the primary key of a direct-access file has the same characteristics as the primary key of an indexed-sequential file. You specify whether the primary key is embedded or nonembedded, its position (if the key is embedded), and the key length. However, the key type for a direct access file is always uncollated; a key type specified for a direct-access file is ignored.

Unlike an indexed-sequential file, sequential access calls to a direct access file while the primary-key is selected do not return the file records sorted by primary-key value. The calls return records according to their physical location in the direct-access file. Records within each block are ordered according to the default ASCII collating sequence, but the blocks are not ordered by primary-key values.

Direct access file records can be accessed in order if one or more alternate keys are defined for the file. The alternate index keeps the alternate-key values in sorted order. Sequential access calls while an alternate key is selected return records in the order provided by the alternate index.

If appropriate, you could define an alternate key for the same field as an embedded primary key. In this way, you could access direct-access file records in primary-key value order.

FORTRAN Keyed-File Interface Concepts

The keyed-file interface is a group of subprogram calls that use the NOS/VE keyed-file interface to operate on keyed files.

How to Use Keyed Files in FORTRAN Programs

This section describes how to use the keyed-file interface in FORTRAN programs. You can also use the keyed-file interface through NOS/VE. See the manual NOS/VE Advanced File Management Usage for more information. You can also use the keyed-file interface through any language, such as COBOL, that uses the standard calling sequence.

Do not use more than one I/O method to process the same file. In particular, do not process the same file using both language statements and keyed-file interface calls.

If you have used CYBER 170 Advanced Access Methods Version 2 (AAM 2), you may want to read about the differences between NOS/VE AAM 2 and the NOS/VE keyed-file interface. These differences are described in appendix E, Differences Between NOS/VE FORTRAN and FORTRAN 5.

Using File Information Tables

The File Information Table (FIT) is a table of values maintained by the FORTRAN keyed-file interface. The values represent file attributes for an instance of open of a file. The FORTRAN keyed-file interface uses values in a FIT to determine how to process a keyed file.

To use a keyed file in your FORTRAN program, first call the FILEIS or FILEDA procedure to create a FIT for the file. (FILEIS for an indexed-sequential file; FILEDA for a direct-access file.) NOS/VE allocates system space for the File Information Table; your program does not need to reserve space for it.

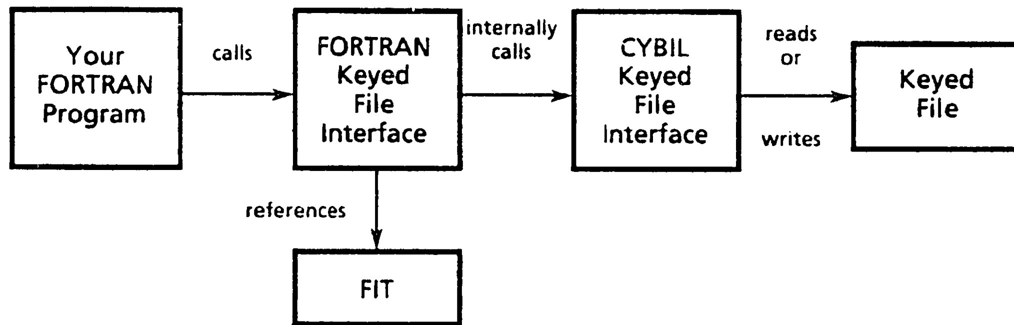
Specify the file's attributes with pairs of FIT keywords and FIT values. For example, this FILEIS call creates a FIT for an indexed-sequential file with a local file name of NEW_IS_FILE, a key length of 5, a maximum record length of 80, and a minimum record length of 5.

```
      CALL FILEIS (fit_ptr,
     #           '$LOCAL_FILE_NAME',         'new_is_file',
     #           '$KEY_LENGTH',              5,
     #           '$MAXIMUM_RECORD_LENGTH',   80,
     #           '$MINIMUM_RECORD_LENGTH',   5)
```

The FILEIS or FILEDA call stores a pointer to the FIT in a variable you specify on the call. Each subsequent keyed-file interface call for the file specifies the FIT pointer variable as its first parameter.

You can set FIT values using STOREF calls and fetch FIT values using IFETCH calls. FIT values are described in detail in chapter 7, File Information Table Keywords and Values.

This figure illustrates how your program can access keyed-file data.



Creating a Keyed File

A FORTRAN program to create a keyed file must perform these steps using the indicated keyed-file interface call:

1. Create a FIT with FILEIS or FILEDA.
2. Open the file with OPENM.
3. Optionally, write records to the file with PUT.
4. Close the file with CLOSEM.

Keyed-file interface calls are described individually in chapter 6, Keyed-File Interface Calls.

You specify the keyed-file attribute values before opening the new keyed file. The file attributes can be specified by one or more of the following:

- The FILEIS or FILEDA call that creates the FIT for the file
- One or more STOREF calls after the FILEIS or FILEDA call
- One or more NOS/VE SET_FILE_ATTRIBUTE commands executed before the program creating the file is executed. (Values specified by a SET_FILE_ATTRIBUTE command override values specified by FILEIS, FILEDA, and STOREF calls.)

If you do not specify a keyed-file attribute by one of these means, a default value is used when the file is opened.

Keyed-File Attributes

You can specify keyed-file attributes as FIT values. The individual FIT value descriptions are in chapter 7, File Information Table Keywords and Values. The keyed-file attributes are as follows:

- File organization attribute:

  $FILE_ORGANIZATION

- Record attributes:

  $RECORD_TYPE (default is undefined)
  $MAXIMUM_RECORD_LENGTH (required)
  $MINIMUM_RECORD_LENGTH (recommended if the record length is variable)

- Primary-key attributes:

  $EMBEDDED_KEY (default is embedded)
  $KEY_LENGTH (required)
  $KEY_POSITION (default is 0)
  $KEY_TYPE (default is uncollated)
  $COLLATE_TABLE_NAME (required if the key type is collated)

- File structure attributes:

  $RECORD_LIMIT
  $MAXIMUM_BLOCK_LENGTH

- Indexed-sequential structure attributes:

  $DATA_PADDING (default is 0%)
  $INDEX_PADDING (default is 0%)

- Direct access structure attributes:

  $INITIAL_HOME_BLOCK_COUNT
  $HASHING_PROCEDURE_NAME

- Block-length guideline attributes (specify instead of $MAXIMUM_BLOCK_LENGTH):

  $AVERAGE_RECORD_LENGTH
  $ESTIMATED_RECORD_COUNT
  $INDEX_LEVELS
  $RECORDS_PER_BLOCK

- Processing attributes:

  $COMPRESSION_PROCEDURE_NAME
  $ERROR_LIMIT (default is 0)
  $LOCK_EXPIRATION_TIME (default is 60,000 milliseconds)
  $MESSAGE_CONTROL (default is only fatal and catastrophic error messages)

- Recovery attributes:

  $FORCED_WRITE (default is unforced)
  $LOG_RESIDENCE (default is none)
  $LOGGING_OPTIONS (default is none)

The keyed-file attributes are described in the NOS/VE Advanced File Management Usage manual. The complete NOS/VE SET_FILE_ATTRIBUTES command description is in the NOS/VE Commands and Functions manual.

NOTE

Besides the required keyed-file attributes, a FORTRAN program must also set the $LOCAL_FILE_NAME value in the FIT. If the $LOCAL_FILE_NAME value has not been specified, the OPENM call returns a fatal error.

Using an Existing Keyed File

A FORTRAN program to process an existing keyed file must perform these steps using the indicated keyed-file interface call:

1. Create a FIT with FILEIS or FILEDA.
2. Open the file with OPENM.
3. Perform the intended operations on the file (described next).
4. Close the file with CLOSEM.

Only temporary file attributes can be specified for an existing keyed file. Preserved file attributes are stored with the file and copied to the FIT by the OPENM call.

The calls listed in parentheses in this list are described individually in chapter 6, Keyed-File Interface Calls.

These operations can be performed on an open keyed file:

- Fetch and store FIT values (IFETCH, STOREF).
- Position the file (REWND, SKIP, STARTM).
- Read records (GET, GETN).
- Write records (PUT, PUTREP).
- Replace records (REPLC, PUTREP).
- Delete records (DLTE).
- Flush modified file blocks to disk (FLUSHM).
- Request locks (LOCKF, LOCKK).
- Clear locks (UNLOCKF, UNLOCKK).
- Use parcels when updating records (PBEGIN, PCOMMIT, PABORT, PDETERM).
- Create alternate keys (RMKDEF).
- Select keys and nested files (STOREF).
- Fetch alternate key information (KLCOUNT, KEYLIST, KLSPACE).
- Build and use result sets to read records (RSBUILD, RSCLEAR, RSCLOSE, RSCOMB, RSDLTE, RSGETN, RSINFO, RSOPEN, RSPUT, RSREWND, RSSKIP, and RSSTART).

Processing Errors

When a keyed-file interface call (other than the FILEIS or FILEDA call) detects an error, it performs these steps:

1. Sets the $ERROR_STATUS value in the FIT to the status condition code of the error.

2. Sets the fatal/nonfatal (FNF) flag in the FIT to indicate whether the severity of the error is fatal or nonfatal.

3. Writes the error message to the $ERRORS file (if indicated by the $MESSAGE_CONTROL value).

   (If the status severity is warning or informational, the keyed-file interface performs only step 3, writing the message.)

4. For a nonfatal error, it increments the $ERROR_COUNT value in the FIT and compares the $ERROR_COUNT value and the $ERROR_LIMIT value.

   If it finds that the $ERROR_COUNT value is equal to the $ERROR_LIMIT value, it changes the $ERROR_STATUS value to the fatal error code AA3255 (error limit reached) and processes the new error (starting at step 2).

5. If an error-exit procedure is specified in the FIT, it calls the procedure.

   The error-exit procedure should fetch the FNF flag to determine if the error is fatal. If the error is fatal, it should close the file because further file processing is not allowed after a fatal error. (Any calls, except CLOSEM or FLUSHM, issued after a fatal error cause a catastrophic error.)

In general, nonfatal errors detected by the keyed-file interface calls are returned in the status variable defined by the STATUS parameter. However, there are two exceptions:

1. If the trivial error limit is exceeded, the status variable contains the "trivial error limit exceeded" error, rather than the error that caused the condition.

2. There are some conditions that permit the keyed-file interface to continue detecting errors. In these cases, the last error detected is the one returned in the status variable. These conditions are:

   aae$enable_altkey_duplicates
   aae$sparse_key_beyond_eor_first
   aae$msg_key_delimiter_first

A FORTRAN program can specify an error-exit procedure by these methods:

- By specifying the error-exit procedure as the $ERROR_EXIT_NAME value before the file is opened.
- By specifying the error-exit procedure as the $ERROR_EXIT_PROCEDURE_NAME value (before or after the file is opened).
- By specifying the error-exit procedure as a parameter on a keyed-file interface call.

If the error-exit procedure is specified by the $ERROR_EXIT_NAME value, it becomes effective only when the file is opened. Otherwise, the error-exit procedure becomes effective when it is specified.

If no error-exit procedure has been specified, the keyed-file interface does not call an error-exit procedure when it detects an error. It stores the $ERROR_STATUS value in the FIT, but the program must check the $ERROR_STATUS value after each call.

To check for an error, the program calls IFETCH to check the $ERROR_STATUS value. If IFETCH returns a nonzero value, it indicates that the call did not complete successfully, and the program should take the appropriate action.

The error-exit procedure or the program can fetch the FNF value from the FIT to determine if the error severity was fatal or nonfatal. It can also use the $ERROR_STATUS value to determine the exact status condition returned.

In one instance, the keyed-file interface clears the $ERROR_STATUS value when it returns from an error-exit procedure.

If a call specifies a working storage area, key area, or primary-key area that is not in a common block, the keyed-file interface detects the error and begins the error processing steps described earlier.

It writes an error message to the $ERRORS file (if requested by the $MESSAGE_CONTROL value) and calls the error-exit procedure (if one is specified in the FIT). If the error-exit procedure fetches the $ERROR_STATUS value, IFETCH returns the value AA2535.

However, unlike other errors, when it finishes processing this error, the keyed-file interface clears the $ERROR_STATUS value so that the get or put operation can complete.

Alternate Keys 2

What Are Alternate Keys? 2-1
Alternate-Key Characteristics 2-1
The Alternate Index 2-2
Alternate-Key Definition 2-2
Duplicate Key Values 2-3
Duplicate-Key Value Error Processing 2-4
Null Suppression 2-4
Sparse-Key Control 2-5
Concatenated Keys 2-6
Repeating Groups 2-7
Variable-Length Keys 2-8
Attributes Incompatible With Variable-Length Keys 2-12
Using a Variable-Length Key 2-12
Nested Files 2-13

How to Use Alternate Keys in FORTRAN Programs 2-15
Alternate Key Creation 2-15
Alternate-Key Use 2-15
Selecting a Key 2-15
Key Selection by Name 2-16
Specifying an Alternate-Key Value 2-16
Key Values Returned 2-17
Collated Key Values 2-17
Fetching Information From the Alternate Index 2-18

Alternate Keys 2

What Are Alternate Keys?

A record within a keyed file can always be accessed by its primary-key value. An alternate key provides an additional way to access records.

An alternate key defines a value in the data record by which the record can be accessed. An alternate key is defined as a field or group of fields in the record.

Although a program can use alternate keys to read records or to position a file, alternate keys cannot be used to write, replace, or delete records. The primary-key value must be used to identify a record to be written, replaced, or deleted.

Alternate-Key Characteristics

Alternate-key fields can overlap each other and the primary key. For example, the primary-key field could be bytes 0 through 9 and two alternate-key fields bytes 0 through 19 and bytes 4 through 14.

Unlike a primary-key value, one alternate-key value can be associated with several records in a file. This is because an alternate-key value does not need to be unique. The same alternate-key value can occur in several records. For example, the same job title can be associated with many names as follows:

Table 2-1. Data Records With Duplicate Alternate-Key Values

Record Number	Primary-Key Field	Alternate-Key Field
1	Hanson	Computer Programmer
2	Jones	Computer Programmer
3	Smith	Computer Programmer

A record can contain more than one alternate-key value if the alternate key is defined as a field that repeats in the record; thus, a single record could contain several alternate-key values. For example, the license numbers of several cars owned by one person as follows:

Table 2-2. Data Record With Several Alternate Keys

Record Number	Primary-Key Field	Alternate-Key Field 1	Alternate-Key Field 2	Alternate-Key Field 3
1	R. Petty	1 LB AU	2ASM451	ELK 592

The Alternate Index

The index for the primary key was described in chapter 1, Keyed-File Interface Concepts. Each alternate key defined for a file has its own index.

An alternate index contains index records, each of which associates an alternate-key value with the primary-key values of the records containing that alternate-key value. The list of primary-key values associated with an alternate-key value is the key list for that alternate-key value.

When you select an alternate key and then specify an alternate-key value, the system searches for the value in the alternate index. If it finds the alternate-key value, it uses the primary-key values in the key list for the alternate-key value to access the data records.

For Better Performance

When one or more alternate keys are defined for a file, file updates require more time because the alternate indexes must also be updated. Alternate keys should be used only when the additional record access capability offsets the cost of increased time spent for file updates.

Alternate-Key Definition

The attributes of an alternate key are specified by its alternate-key definition.

These attributes are required to define an alternate key:

- Key name
- Key position
- Key length

An alternate key has a name so that it can be selected for use. The alternate-key position and length define the alternate-key field within the record.

These optional attributes define how the alternate key is processed:

- Key type
- Collate table name (if the key type is collated)
- Duplicate key values
- Null suppression
- Sparse-key control
- Repeating groups
- Concatenated key
- Variable-length key

The key type of an alternate key determines the order of the alternate-key values in the alternate index, and therefore, the order in which records are accessed sequentially when you use the alternate key. The key types for an alternate key are the same as the key types for the primary key as described in chapter 1, Keyed-File Interface Concepts.

If the key type is collated, you can explicitly specify a collation table for the alternate key or use, as the default, the collation table for the primary key (if one has been specified).

Duplicate Key Values

By default, duplicate values for an alternate key are not allowed. However, if you want to allow duplicate key values, you can specify whether the records having the same alternate-key value are accessed in primary-key-value order or in first-in-first-out order.

In a key list ordered by primary-key value, the primary-key values are stored in sorted order according to the primary-key type. New values are added to the key list so that the primary-key-value order is kept.

In a key list ordered first-in-first-out, the primary-key values are stored in the key list in the order the values are added to the key list, instead of in primary-key-value order. New values are always added to the end of the key list.

For Better Performance

When alternate-key values are frequently duplicated in a file, the key lists should be ordered by primary-key value. First-in-first-out ordering of key lists requires that delete and replace operations sequentially search the key list to find the primary-key value of the updated record; a sorted key list provides faster access to a primary-key value.

For example, suppose you write three records to the file in this order:

Record Number	Primary-Key Field	Alternate-Key Field
1	McDarrels	Hamburgers
2	Burger Duke	Hamburgers
3	Willys	Hamburgers

The following shows the resulting key list in primary-key order and in first-in-first-out order:

Table 2-3. Ordered by Primary Key

Record Number	Primary-Key Field	Alternate-Key Field
2	Burger Duke	Hamburgers
1	McDarrels	Hamburgers
3	Willys	Hamburgers

Table 2-4. Ordered by First-In-First-Out

Record Number	Primary-Key Field	Alternate-Key Field
1	McDarrels	Hamburgers
2	Burger Duke	Hamburgers
3	Willys	Hamburgers

Duplicate-Key Value Error Processing

If duplicate values are not allowed and a duplicate is found in a record about to be written to the file, the record is not written to the file and a nonfatal error (status AA2100) is returned.

A nonfatal error (status AA2865) also occurs if a duplicate value is found while a new alternate index is being created. However, the record containing the duplicate value cannot be discarded, as it is already in the file. Subsequent processing depends on whether incrementing the nonfatal-error count causes the count to exceed the nonfatal-error limit a set by the user.

- If the nonfatal-error limit is not exceeded, the apply operation redefines the alternate key to allow duplicates, ordered by primary-key value, discards the partially built index, and builds the redefined index.
- If the nonfatal-error limit is reached, the apply operation returns AA2870 and removes all alternate indexes it has created. (Deleted indexes are not restored.)

In either case, a message describing the action taken is written to the $ERRORS file.

Null Suppression

By default, if an alternate-key field contains a null value, the null value is stored as the alternate-key value for the record. The null_suppression file attribute allows you to exclude null values from an alternate index.

Null suppression excludes any record with a null alternate-key value from the alternate index. Null suppression can save space, access time, and update time because the index is smaller when null alternate-key values are excluded. (Null suppression does not remove the null value from the data record.)

The null value depends on the key type as follows:

Key Type	Null Value
Integer	Zero
Uncollated	Spaces
Collated	Spaces (before collation)

If null suppression is not specified, records containing a null value in the alternate-key field are indexed by the null value. The records can later be accessed by specifying the null value as the alternate-key value.

For example, suppose a membership file has an alternate key of the spouse's name. Unmarried members would have a null value for the alternate-key field. Therefore, the key list for the null value lists all unmarried members. The following shows the alternate index with and without null suppression:

Without Null Suppression

Spouse's Name	Member's ID
	1626736
	8273648
Diana Simmons	4872672
Mark Ramsey	7726184
Shelly Gable	2673651

With Null Suppression

Spouse's Name	Member's ID
Diana Simmons	4872672
Mark Ramsey	2673651
Shelly Gable	7726184

Sparse-Key Control

You can use sparse-key control to create an alternate index that includes or excludes records depending on the character in a specific position in the record.

For example, suppose a student file has a one-character code indicating the student's class. To get a mailing list for juniors and seniors only, you could define an alternate index controlled by the class code.

To specify sparse-key control, you specify three values:

Value	Example
Sparse-key control position	Position of the class code in the record
Sparse-key control characters	Junior and senior class code characters
Sparse-key control effect (Indicates whether the alternate-key value should be included or excluded if the sparse-key character matches)	Included if the class code indicates a junior or senior record

Assume that the sparse-key control position is the first character after the name field and that the junior and senior class codes are 3 and 4. If the following records are copied to the file, the first three records are included in the alternate index, but not the last record.

```
Louis Skolnik     4
Gilbert Sullivan  4
Elliot Wermzer    3
Judy Manhasset    2
```

The sparse-key control position must be within the minimum record length. If you specify sparse-key control for an alternate key, the alternate-key field or fields need not be within the minimum record length.

A nonfatal (trivial) error (status AA2875) is returned if both of these conditions are true for a record:

- The character at the sparse_key_control_position indicates that the record should be included in the alternate index.
- The record has no alternate-key value because the record is too short to contain the entire alternate-key value.

When an apply or write operation detects this error, it does not include the record in the alternate index. (A write operation does write the record to the file.)

Concatenated Keys

A concatenated key is an alternate key formed from several fields, or pieces, in the record. A concatenated key can comprise up to 64 pieces.

The concatenated pieces can be noncontiguous and can be concatenated in any order. Each piece can be a different key type. All collated-key pieces use the same collation table.

To create a concatenated key in a FORTRAN program, use the SCLCMD call to execute the CREATE_ALTERNATE_INDEXES utility. The CREATE_ALTERNATE_INDEXES utility is described in the NOS/VE Advanced File Management Usage manual.

The first piece you specify is the leftmost piece of the key. You specify it the same as you specify a nonconcatenated key. The pieces to be concatenated to the leftmost field are defined by individual ADD_PIECE subcommands. The subcommand order specifies the order of the concatenated pieces.

A concatenated key can use sparse-key control and/or null suppression. A concatenated key is considered to have a null value if the values in all fields of the key are null (before collation for collated keys).

For example, suppose you decide to define an alternate key consisting of the initials of the member's name. The first piece of the key value would be the first letter of the member's first name, the second piece would be the first letter of the member's middle name, and the third piece would be the first letter of the member's last name. Consider this data record:

0	20	40
Kennedy	John	Fitzgerald

The alternate-key value is JFK, assuming the concatenated-key pieces are defined as:

First piece: `Key_Position=20, Key_Length=1`

Second piece: `Key_Position=40, Key_Length=1`

Third piece: `Key_Position=0, Key_Length=1`

Repeating Groups

The repeating_groups file attribute allows a data record to contain more than one value for the same alternate key. This allows a primary-key value to be associated with more than one alternate-key value.

To specify an alternate-key field within a repeating group:

1. Specify the first alternate-key field by its key position, key length, and key type. All subsequent alternate-key fields have the same length and type as the first.

2. Specify repeating groups for the alternate key by specifying the repeating group length, that is, the distance from the beginning of the first instance of the alternate key to the beginning of the second instance of the alternate key in the record.

3. Specify the repeating-group count, which is the number of times the alternate-key field repeats in the record.

You can specify that the repeating group repeats a fixed number of times or that it repeats until the end of the record.

- If the alternate-key field repeats a fixed number of times, all alternate-key fields must be within the minimum record length.

- If the alternate-key field repeats to the end of the record, the minimum record length imposes no restriction. The system stores as many alternate-key values as the record length allows.

Repeating groups cannot be used with concatenated keys or when duplicate-key values are allowed and ordered first-in-first-out.

For example, suppose each record in a membership file lists the sports the member enjoys and his years of experience as follows (columns are counted from zero):

Field:	Sports and Sports Experience
Columns:	Variable number of 2-field pairs beginning at column 75
	The Sports field is 10 characters followed by a 2-digit Sports Experience field
Type:	ASCII characters

You could define an alternate key for the Sports values (without the Sports-Experience values) as follows:

CREATE_KEY_DEFINITION parameters:

```
Key_Position=75, Key_Length=10, Key_Type=uncollated, Repeating_Group_Length=-
12,
Repeating_Group_Count=repeat_to_end_record,
Duplicate_Key_Values=ordered_by_primary_key
```

RMKDEF call:

```
CALL RMKDEF(fit, 0, 75, 10, 0, 'UNCOLLATED', 'ORDERED_BY_PRIMARY_KEY', 12, 0)
```

The key list for an alternate-key value would list the identification numbers of all members that enjoy that sport.

The following shows the primary keys for three records and their contents from column 75 to the end of the record:

Primary Key	Record Contents Beginning at Column 75
1662876	Volleyball02Running 03Basketball02
6166287	Bicycling 10Volleyball01
0027840	Running 15Running 15Running 15

If these were the only records in the file, the alternate index would appear as follows:

Alternate Key Value	Primary Key Value
Basketball	1662876
Bicycling	6166287
Running	0027840 1662876
Volleyball	1662876 6166287

Notice that the key type is the default (uncollated) and the duplicate-key values specification is ordered_by_primary_key. Thus, each key list is sorted according to the default ASCII collating sequence.

Notice also, as shown by the Running key list, each primary-key value is listed only once in a key list, regardless of the number of times the alternate-key value occurs in the record.

Variable-Length Keys

A variable-length alternate key is an alternate key whose value varies in length. The definition of a variable-length alternate key specifies the key's starting position, maximum length, and set of delimiter characters.

The end of a variable-length key value is marked by a delimiter character, the end of the key field, or the end of the record, whichever is found first starting at the key_ position.

By defining the key as a variable-length key, you can use the following values as alternate keys:

- The first value beginning at a certain position of each record.
- The last field in a variable-length record.
- All data in a variable-length record.

By defining the key as a variable-length key with the repeating groups attribute, you can use the following values as alternate keys:

- A value found anywhere in a fixed-length field (if all other characters in the field are in the set of delimiter characters for the alternate key).

- Each value in a sequence of values, separated by one or more consecutive delimiter characters. The sequence of values can be within:
  - A fixed-length field
  - A variable-length field at the end of the record
  - The entire record

For Better Performance

Define a key as a variable-length key only when necessary. The requirement to scan the key field for delimiter characters adds processing time when the alternate index is built and when the file is updated.

The following examples each specify a variable-length alternate key.

Example 1:

The alternate key is to be the first sequence of up to 80 non-blank characters in each record.

0 EOR

First token in each record.

Key Value

To define the alternate key, specify the key position as 0, the key length as 80, and the variable-length key attribute with the blank character as the delimiter, as follows:

CREATE_KEY_DEFINITION parameters:

```
Key_Position=0, Key_Length=80, Variable_Length_Key=' '
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,0,80,0,0,0,0,0,0,0,0,0,' ')
```

Example 2:

Assume that each record consists of a required 20-byte portion followed by an optional variable-length portion of up to 120 bytes.

0 20 EOR

Fixed portion : Variable portion

Key Value

To define the variable-length portion as the alternate key, specify the key position as 20, the key length as 120, and the variable-length key attribute with an empty delimiter set.

The statements to define the key are the same as for example 1 except for the following:

CREATE_KEY_DEFINITION parameters:

```
Key_Position=20, Key_Length=120, Variable_Length_Key=''
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,20,120,0,0,0,0,0,0,0,0,0,'')
```

Example 3:

Assume a 100-byte field that contains a value at byte 5 that is the alternate key. The value is from 0 through 95 bytes long, right-justified and blank-filled within the field.

0 5 99

right-justified

Key Value

To define the alternate key, specify the key position as 5, the key length as 95, the variable-length key attribute with the blank character as the delimiter, and the repeating_groups attribute.

The repeating_groups attribute is required because the value is right-justified in the field; thus, the search for the value must not end at the first delimiter; it should continue to the end of the field. For a repeating variable-length key, the repeating_group_length value can be any integer greater than zero; the repeating_group_count is the length of the alternate-key field.

CREATE_KEY_DEFINITION parameters:

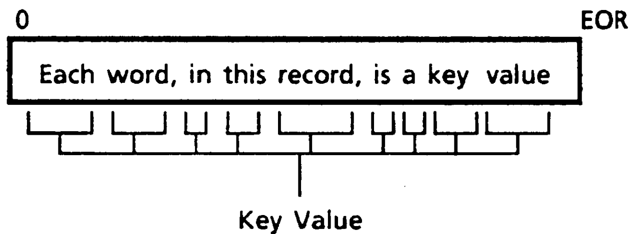
```
Key_Position=5, Key_Length=95, Variable_Length_Key=' ',
Repeating_Group_Length=1, Repeating_Group_Count=95
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,5,95,0,0,0,1,95,0,0,0,0,' ')
```

Example 4:

Each string of letters in the data is to be defined as a value for the alternate key.



To define the alternate key, specify the key position as 0, the key length as the maximum record length (80), the variable-length key attribute, and the repeating_groups attribute. Notice that the delimiter set is defined as all characters except the letters.

CREATE_KEY_DEFINITION commands:

```
/var
var/key_delimiters: string = ..
var../' 1234567890-=!@#$%^&*()_+[]`{}";''\:"!,./<>?'..
var../$CHAR(000)//$CHAR(001)//$CHAR(002)//$CHAR(003)// ..
var../$CHAR(004)//$CHAR(005)//$CHAR(006)//$CHAR(007)// ..
var../$CHAR(008)//$CHAR(009)//$CHAR(010)//$CHAR(011)// ..
var../$CHAR(012)//$CHAR(013)//$CHAR(014)//$CHAR(015)// ..
var../$CHAR(016)//$CHAR(017)//$CHAR(018)//$CHAR(019)// ..
var../$CHAR(020)//$CHAR(021)//$CHAR(022)//$CHAR(023)// ..
var../$CHAR(024)//$CHAR(025)//$CHAR(026)//$CHAR(027)// ..
var../$CHAR(028)//$CHAR(029)//$CHAR(030)//$CHAR(031)// ..
var../$CHAR(127); varend

  create_key_definition, key_name=words, ..
    key_position=0, key_length=80, ..
    variable_length_key=key_delimiters, ..
    repeating_group_length=1, ..
    repeating_group_count=repeat_to_end_of_record
```

RMKDEF Call:

```
    value=' 1234567890-=!@#$%^&*()_+[]`{}";''\:"!,./<>?'
+ //$CHAR(000)//$CHAR(001)//$CHAR(002)//$CHAR(003)
+ //$CHAR(004)//$CHAR(005)//$CHAR(006)//$CHAR(007)
+ //$CHAR(008)//$CHAR(009)//$CHAR(010)//$CHAR(011)
+ //$CHAR(012)//$CHAR(013)//$CHAR(014)//$CHAR(015)
+ //$CHAR(016)//$CHAR(017)//$CHAR(018)//$CHAR(019)
+ //$CHAR(020)//$CHAR(021)//$CHAR(022)//$CHAR(023)
+ //$CHAR(024)//$CHAR(025)//$CHAR(026)//$CHAR(027)
+ //$CHAR(028)//$CHAR(029)//$CHAR(030)//$CHAR(031)
+ //$CHAR(127)
  CALL RMKDEF(fit,0,0,80,0,0,0,1,0,0,0,0,0,value)
```

Attributes Incompatible With Variable-Length Keys

The following alternate-key attributes are not supported for variable-length keys:

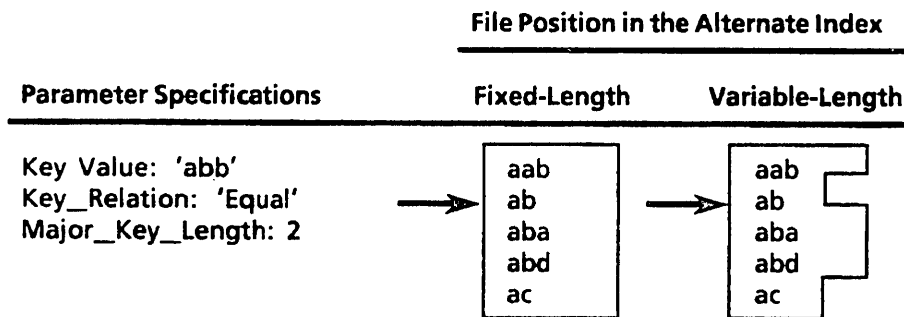
- Integer key type
- Ordering duplicate-key values chronologically (First_In_First_Out)
- Concatenation
- Null suppression
- Sparse-key control

Using a Variable-Length Key

Using a variable-length alternate key differs from using a fixed-length key in the following ways:

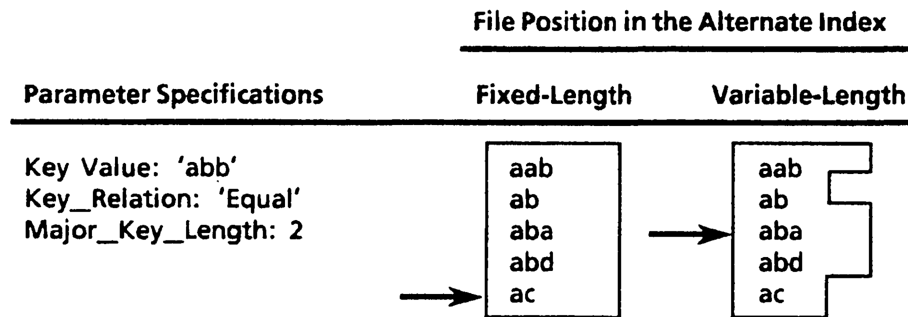
- On a call using a variable-length key, you must specify the length of the key value as well as its location. The length of the key value is specified using the appropriate major-key length parameter.
- When a call returns a variable-length key value, it returns the value padded with delimiter characters to the full key length. (It pads using the character with the lowest ASCII value in delimiter set.)
- The key value specified on the call is compared with the full key value stored in the index, not only the leftmost bytes.

Key value comparison is illustrated by the following example that contrasts the use of a variable-length key value with the use of an equivalent major-key value for a fixed-length key. The key value used is the leftmost two bytes ('ab'):



As shown, when the Key_Relation is 'Equal', the positioning is the same.

However, the positioning can differ if the Key_Relation is 'Greater_Than':



The file positioning differs because:

- The two-byte major-key value is compared with the leftmost two bytes of the fixed-length alternate-key values. So, the file is positioned at the first key value whose leftmost two bytes are greater than 'ab', that is, 'ac'.
- The two-byte variable-length key value is compared with the full variable-length alternate-key value, not just the leftmost two bytes. So, the file is positioned at the first key value greater than 'ab', that is, 'aba'.

Nested Files

A nested file is a file structure defined within a NOS/VE file cycle. It is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

The keyed-file interface provides nested files to extend the NOS/VE limit on the number of files a task can use. All nested files defined in a file share the same memory segment. This provides effective memory use when the nested files are much smaller than the segment size limit (2^{32} bytes).

All nested files in a file share the same NOS/VE catalog entry. Thus, if one nested file is damaged, the entire file is damaged and requires recovery.

The keyed-file interface creates the initial nested file (named $MAIN_FILE) when it creates the keyed file. It uses $MAIN_FILE as the default nested file; other nested files are used only when explicitly selected.

No FORTRAN keyed-file interface call exists to create a nested file. However, a FORTRAN program can create a nested file (other than $MAIN_FILE) by:

- Calling the CYBIL subprogram AMP$CREATE_NESTED_FILE
- Calling the NOS/VE command COPY_KEYED_FILE
- Copy an existing nested file
- Calling the CREATE_KEYED_FILE utility

(The AMP$CREATE_NESTED_FILE call is described in the CYBIL Keyed-File and Sort/Merge Interfaces manual. The COPY_KEYED_FILE command and the CREATE_KEYED_FILE utility are described in the NOS/VE Advanced File Management manual.)

A FORTRAN program selects a nested file by storing its name in the FIT using the keyword $NESTED_FILE_NAME (or $NFN). To re-select the default nested file, it stores the name $MAIN_FILE.

Each alternate-key definition applies to only one nested file. To define an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then define the alternate key. Similarly, to select an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then select the alternate key.

A task can perform operations only on the currently selected nested file. However, the file position, key selection, and locks for a nested file are not lost when another nested file is selected. For example, consider this sequence of events:

1. A task is issuing GETN calls while NESTED_FILE_1 and ALTERNATE_KEY_1 are selected.

2. The task selects and uses NESTED_FILE_2.

3. The task selects NESTED_FILE_1 again. It can continue reading records sequentially from the file position at which it stopped reading when it selected NESTED_FILE_2. The same key, ALTERNATE_KEY_1, remains selected.

How to Use Alternate Keys in FORTRAN Programs

Alternate Key Creation

The recommended method for creating alternate keys is to use the NOS/VE utility CREATE_ALTERNATE_INDEXES. In general, using the utility is easier and more efficient than writing a program especially when creating more than one alternate key.

You can execute the utility from a FORTRAN program using the SCLCMD call. The SCLCMD call is described in chapter 6, Keyed-File Interface. CREATE_ALTERNATE_INDEXES is described in the NOS/VE Advanced File Management Usage manual.

The RMKDEF call both defines the alternate key and applies the definition to the keyed file to build the alternate index.

The RMKDEF call can be issued for a keyed file that has been created before or during program execution. Both a FILEIS (or FILEDA) call and an OPENM call must be executed before the RMKDEF call. The RMKDEF call uses the FIT pointer returned by the FILEIS (or FILEDA) call.

The alternate key created by the RMKDEF call remains as part of the keyed file for the life of the file or until the alternate key is explicitly deleted. You can delete an alternate key using the NOS/VE utility CREATE_ALTERNATE_INDEXES.

Alternate-Key Use

You can use an alternate key to position or read a keyed file. (Calls to write to a keyed file must specify primary-key values, not alternate-key values.)

While an alternate key is selected, the file is positioned and records are read in the logical record order defined by the alternate index. For example, each GETN call reads the next record in alternate-key order, instead of in primary-key order.

Selecting a Key

To indicate that the key values on subsequent STARTM, GET, and GETN calls are alternate-key values, you must call STOREF to select the alternate key. The key selection takes effect when the next START, REWND, or GET (but not GETN) call is issued.

Use the STOREF call to specify a key by its name.

Key Selection by Name

The STOREF call can select a key by storing the key name in the FIT.

For example, the following STOREF call selects alternate key ALTERNATE_567_9_250.

```
CALL STOREF(fit,'$KEY_NAME','ALTERNATE_567_9_250')
```

To change the key selection, you call STOREF again, specifying another alternate key or the primary key. The primary key name is $PRIMARY_KEY. For example, the following call selects the primary key:

```
CALL STOREF(fit,'$KEY_NAME','$PRIMARY_KEY')
```

Selection by key name is the only way to select a nonembedded primary key.

Specifying an Alternate-Key Value

You can specify an alternate-key value in the working storage area.

If you specify the value in the working storage area, you must store the value in the alternate-key position in the working storage area. If the alternate key is a concatenated key, each piece must be stored in its field in the record.

For example, suppose you define your working storage area as an 80-integer array named WSA. If the alternate-key field is the fifth integer (that is, the alternate-key field begins at byte 32 [counting from zero] and is 8 bytes long), you could store the integer alternate-key value 1374 as follows:

```
WSA(5)=1374
```

The file-position values returned, and their meanings, differ when using an alternate key, instead of the primary key, as follows:

$FILE_ POSITION Value	Meaning
1	The file is positioned at the beginning of the alternate index. (It is positioned to read the record with the lowest alternate-key value.)
8	The file is positioned at the end of the key list for the current alternate-key value. (It is positioned to read the first record having the next alternate-key value.)
16	The file is positioned at the end of a record, but not at the end of the key list. (It is positioned to read the next record having the current alternate-key value.)
64	The file is positioned at the end of the alternate index. (It cannot read a record at this position.)

When reading a file sequentially, you should call IFETCH to fetch the file position and then check the returned value after each get call.

To get all records containing the same alternate-key value, the program issues GETN calls until a file position of 8 (end-of-key-list) is returned.

When a GET or GETN call returns a file position of 64, it has positioned the file at its end-of-information and no GETN calls should be issued until the file is repositioned.

A GETN call issued after a call that positions the file at the end-of-information is an attempt to read beyond the end-of-information. It returns non-fatal error $ERROR_STATUS value AA2635.

Key Values Returned

You can fetch both the alternate-key value and the primary-key value from the FIT while an alternate key is selected.

- A GETN call issued while an alternate key is selected returns the alternate-key value of the record read in the key area, instead of the primary-key value.
- A GETN (or GET or STARTM) call issued while an alternate key is selected can return the primary-key value of the record read in a primary-key area.

Before a call can return a value in a primary-key area, the program must store in the FIT the location of the primary-key area.

For example, this call specifies the variable PRIKEY as the primary-key area:

```
CALL STOREF(fit,'$PRIMARY_KEY_ADDRESS',prikey)
```

NOTE

Like the key area and the working-storage area, the primary-key area should be in a common block.

The primary-key area is used only while an alternate key is selected; no value is returned in the primary-key area while the primary key is selected.

Collated Key Values

If the key type of the key is COLLATED, the key value returned may no longer be the key value input with the record. This can occur if the collation table assigns the same collation weight to more than one character code.

The process is as follows:

1. Each character of a collated key value is stored in the index as the lowest character code having the same collating weight.
2. When the key value is returned, the key value is decollated to its original form. However, if more than one character code is collated as the same value, the value returned is the lowest character code with the same collation weight.

Because of this process, your program may not be able to fetch a nonembedded primary-key value in its original form. (It can always fetch an alternate-key or embedded primary-key value in its original form from the record data.)

For example, if lowercase letters are collated as equal to the corresponding uppercase letters (each lowercase letter is given the same collating weight as the corresponding uppercase letter), the alternate-key value is returned using only uppercase letters.

As another example, consider the OSV$xxxx collation tables predefined by NOS/VE and listed in appendix C, ASCII Character Set and Collating Weight Tables. These collation tables assign collation weight 0 to all unprintable characters and to the space character. Thus, all unprintable characters and all space characters are returned as the lowest character code value with collation weight 0, which is the NULL character (00 hexadecimal).

Fetching Information From the Alternate Index

Your program can fetch information from the alternate index using the KLCOUNT, KEYLIST, and KLSPACE calls.

- The KLCOUNT call returns the number of primary-key values for a range of alternate-key values in the alternate index.
- The KEYLIST call returns the actual primary-key values for a range of alternate-key values.
- The KLSPACE call returns the alternate-index block count for a range of alternate-key values.

These calls differ from the other keyed-file interface calls in these ways:

- Values must be specified for all parameters. (The valid values are listed in the parameter descriptions.)
- The only values that these calls update in the FIT are the file position, the last operation, and the error status. The calls do not use FIT values as default parameter values.

Sharing Keyed Files 3

When Does Sharing Keyed Files Require Locks? 3-2
When to Specify MODIFY Share Mode 3-2

What Are Access and Share Modes? 3-3
About Access Modes 3-3
About Share Modes 3-4
Using Access and Share Modes to Share a File 3-5
The Relationship Between Access Modes, Share Modes, FIT Keywords, ... 3-5
Three Steps to Set Access and Share Modes 3-6
Two Steps to Set Access and Share Modes 3-6
Specifying an Open_option Parameter on the OPENM Call 3-7

What Are Locks? 3-9
Reasons for Locks 3-9
Lock Intents 3-11
Exclusive_Access Lock Intent 3-11
Preserve_Access_and_Content Lock Intent 3-11
Preserve_Content Lock Intent 3-12
Lock Renewal and Lock_Intent Changing 3-12
File Locks 3-12
Waiting for a Lock 3-13
Lock Expiration and Clearing 3-14
How a Lock Expires 3-14
Expired Lock Conditions 3-15
Lock Deadlock 3-16
Lock Conflict Tables 3-16

Sharing Keyed Files 3

A keyed file is shared when it is opened more than once and the occurrences overlap. The file can be opened more than once by the same task, job, or by multiple tasks or jobs. Every time the file is opened by an entity, the period of time during which it is open is called an *instance of open.*

NOTE

The period of time during which a file is open is called an *instance of open.* It describes a time interval that is initiated and controlled by an entity.

A task can have more than one instance of open of a file that overlap. One example of when a task might want to have concurrent instances of open of a file is when a task wants to, in parallel, read successive records from two or more nested files that in the same keyed file.

When Does Sharing Keyed Files Require Locks?

Sharing occurs when instances of open for a keyed file overlap. Generally, you need to use locks whenever sharing may occur, unless the first instance of open specifies exclusive access to the file so no other instance of open can access the file, even for read access.

To have exclusive access to a file, specify a share mode of NONE. For example, this STOREF call stores an access mode of read and a share mode of NONE in the file information table pointed to by FITPTR:

```
call STOREF(fitptr, '$ACCESS_AND_SHARE_MODES', '((READ),(NONE))')
```

Or, you can specify exclusive access to a file by specifying an open_option parameter on the OPENM call.

For Better Performance

Specify exclusive access to a file whenever possible. When you have exclusive access, there is no need to lock records so keyed file processing is more efficient.

For example, this OPENM call specifies an open_option parameter of 'NEW', so the access modes are set to read and write, and the share mode is set to none:

```
call OPENM(fitptr, 'NEW', 0)
```

For more information on the access and share modes set when specifying an open_option parameter on the OPENM call, see Specifying an Open_option Parameter on the OPENM Call later in this chapter.

When to Specify MODIFY Share Mode

When sharing a keyed file, you may want to specify MODIFY access for other users. MODIFY access allows statistics for the file to be kept, which you can access by using a utility such as DISPLAY_KEYED_FILE_PROPERTIES. The statistics include information such as the number of times the file has been accessed, the number of times information has been replaced in the file, and the number of times information has been deleted from the file.

For more information on DISPLAY_KEYED_FILE_PROPERTIES and file statistics, see Displaying, Copying, and Creating Keyed Files in the Advanced File Management manual.

For Better Performance

For better performance when using a keyed file, check that the share modes allowed are no more than those required. If possible, allow no sharing of the file.

Access and share modes, reasons for using locks, and how to use locks are described in detail in the following pages.

What Are Access and Share Modes?

To share files and use locks effectively, you need to understand access and share modes. Access and share modes determine how you and other users can use a file.

About Access Modes

The *access mode* of a file specifies what you can do with the file. Access modes include read, modify, shorten, append, write, and execute. They are defined as follows:

Access Mode	What You Can Do With the File
Read	Read the data in the file.
Modify	Change the data in the file. Modify access does not allow a file to change its size[1]. Modify access also means that statistics are kept for the file.
Shorten	Delete data from the file.
Append	Add data to the file or change data in the file.
Write	Change data, delete data, and add data to the file. Equivalent to modify, shorten, and append access modes.
Execute	Specify the file in an EXECUTE_TASK command or as an SCL procedure.
All	Equivalent to read, modify, shorten, append, and execute access modes.

[1]Modify access alone is generally insufficient if you are changing data in a file. For example, if you change data that forces a data block split, the file must be able to expand to accomodate the new block.

About Share Modes

The *share mode* of a file specifies what access modes other users of the file must specify if they want to use it at the same time. Share modes are similar to access modes, except they apply only to how other users can use the file. They are defined as follows:

Share Mode	What Other Users Can Do With the File
None	Nothing.
Read	Read the data in the file.
Modify	Change the data in the file. Modify access does not allow a file to change its size[1].
Shorten	Delete data from the file.
Append	Add data to the file or change data in the file.
Write	Change data, delete data, and add data to the file. Equivalent to modify, shorten, and append share modes.
Execute	Specify the file in an EXECUTE_TASK command or as an SCL procedure.
All	Equivalent to read, modify, shorten, append, and execute share modes.

[1]Modify access alone is generally insufficient if you are going to allow others to change data in a file. For example, if another user changes data that forces a data block split, the file must be able to expand to accomodate the new block.

In addition to these access and share modes, there are three special modes you can specify with the File Information Table (FIT) keyword $ACCESS_AND_SHARE_MODES. These modes are:

- permitted_access_modes
- required_share_modes
- determine_from_access_modes

These modes use combinations of the previously discussed modes, depending on specific circumstances. For more information, see $ACCESS_AND_SHARE_MODES in chapter 7, File Information Tables.

Using Access and Share Modes to Share a File

When more than one person uses a file at the same time, their access to the file is restricted by access and share modes specified by previous concurrent users.

In other words, when you try to open a file that other users have already opened, your success at opening the file depends on each user's access and share modes.

In order to be successful, the access and share modes that you specify must satisfy these conditions:

- Your requested access modes must have been specified as share modes by each user who has the file open.
- Your requested share modes cannot restrict the access mode of any user who has the file open.

The Relationship Between Access Modes, Share Modes, FIT Keywords, and File Interface Calls

Some FIT keywords and file interface calls affect the access and share modes of a file. The FIT keywords store and fetch information from a file information table. A file information table is maintained for every instance of open of a file. The keywords that affect access and share modes are:

FIT Keyword	Purpose
$ACCESS_AND_SHARE_MODES	Sets the access and share modes in the file information table.
$ACCESS_MODE	Sets the access modes in the file information table. Also sets the share modes to DETERMINE_FROM_ACCESS_MODES.
$GLOBAL_ACCESS_MODE	Fetches the access modes from the file information table.
$GLOBAL_SHARE_MODE	Fetches the share modes from the file information table.
$OPEN_SHARE_MODE	Sets the share modes in the file information table applicable to other instances of open within the same job. $OPEN_SHARE_MODE is different than share modes of $ACCESS_AND_SHARE_MODES, where the share modes are applicable to any other instance of open.

For more detailed information about these FIT keywords, see chapter 7, File Information Tables.

File interface calls that use access and share modes are:

Call	Purpose
FILEIS	Creates a file information table for an indexed-sequential file and, optionally, initializes values in the table.
FILEDA	Creates a file information table for a direct-access file and, optionally, initializes values in the table.
STOREF	Stores values in a file information table.
IFETCH	Fetches values from a file information table.
OPENM	Opens a keyed file.

For more detailed information about these calls, see chapter 6, Keyed-File Interface Calls.

To set the access and share modes of file, use one of these methods:

Three Steps to Set Access and Share Modes

1. Use the FILEIS or FILEDA call to create a file information table.
2. Use the STOREF call to set the access and share mode values in the table.
3. Use the OPENM call, specifying the open_option parameter as 0, to open the keyed file.

Two Steps to Set Access and Share Modes

1. Use the FILEIS or FILEDA call to create a file information table and to set the access and share mode values in the table.
2. Use the OPENM call, specifying the open_option parameter as 0, to open the keyed file.

You can also use the NOS/VE commands SET_FILE_ATTRIBUTES to set the access and share modes for a file. For more information on using these commands, see NOS/VE File Attributes in chapter 4, Catalog and File Management, in the System Usage manual.

There are two important points to remember:

- The OPENM call redefines the access and share mode values in a file information table unless you specify the open_option parameter as 0.
- Any time you use STOREF to store access and share mode values in a file information table, the values are used the next time the file is opened. If you use STOREF after an OPENM call, the access and share mode values are not used for the current instance of open of the file. However, they are used for the next OPENM call as long as you specify the open_option parameter as 0.

Specifying an Open_option Parameter on the OPENM Call

If you do not specify 0 for the open_option parameter on the OPENM call, the access and share modes for the file are determined by the value for the open_option parameter that you specify as follows:

Sample OPENM Call	Access Modes Set	Share Modes Set
OPENM(fit, 'NEW', 0)	Read, modify, shorten, append	None
OPENM(fit, 'INPUT', 0)	Read	Read
OPEN(fit, 'OUTPUT', 0)	Modify, shorten, append	None
OPENM(fit, 'IO', 0)	Read, modify, shorten, append	None

Here are some examples of specifying access and share modes for a file.

- This example creates a file information table, stores access and share mode values in the table, and opens the file:

```
 integer fitptr                      ! Pointer to the file information
                                     !   table.
 call FILEIS(fitptr,                 ! Create a file information table
+ '$LOCAL_FILE_NAME', 'my_file',     !   for an indexed-sequential file.
+ '$KEY_LENGTH', 15,                 ! Define the key length.
+ '$MAXIMUM_RECORD_LENGTH', 80,      ! Define the max record length.
+ '$MINIMUM_RECORD_LENGTH', 15)      ! Define the min record length.

 call STOREF(fitptr,                 ! Store the access and share modes
+ '$ACCESS_AND_SHARE_MODES',         !   in the file information table.
+ '((READ,EXECUTE),(NONE))')         ! Access modes = read and execute
                                     ! Share modes = none
 call OPENM(fitptr, 0, 0)            ! Open the file.
```

- This example creates a file information table, stores the access and share modes, stores the open share mode, and opens the file.

```
 integer fitptr                      ! Pointer to the file information
                                     !   table.
 call FILEIS(fitptr,                 ! Create a file information table
+ '$LOCAL_FILE_NAME', 'new_file',    !   for an indexed-sequential file.
+ '$KEY_LENGTH', 15,                 ! Define the key length.
+ '$MAXIMUM_RECORD_LENGTH', 80,      ! Define the max record length.
+ '$MINIMUM_RECORD_LENGTH', 15)      ! Define the min record length.
+ '$ACCESS_MODE', '(ALL)',           ! Access modes = all,
                                     !   share mode = none.
+ '$OPEN_SHARE_MODE',                ! Open share modes = read and modify
+     '(READ,MODIFY)'                !   (Open share modes apply only to
                                     !    open requests within the job.)
 call OPENM(fitptr)                  ! Open the file.  By default, the
                                     !    open_option and file_position
                                     !    parameters are 0.
```

- This example stores access and share mode values in a file information table, and then opens a keyed file specifying 'NEW' for the open_option parameter. When the access and share modes are fetched, they have been changed because the open_option parameter did not specify 0.

```
 integer fitptr, mode              ! Pointer to the file information
 .                                 !   table and the mode returned
 .                                 !   from the table
 .
 call STOREF(fitptr,               ! Access modes = all
+ '$AASM', '((ALL),(ALL))')        ! Share modes = all

 call OPENM(fitptr, 'NEW', 0)      ! Open the file as a new file.

 call IFETCH(fitptr,
+ 'GLOBAL_ACCESS_MODE', mode)      ! Mode = 3, which is read, modify,
                                   !   shorten, and append.
 call IFETCH(fitptr,
+ 'GLOBAL_SHARE_MODE', mode)       ! Mode = 0, which is no sharing.
```

- This example shows how to allow the first open request to update the file and to restrict successive open requests to read access. To do this, specify more than one set of access and share modes using the $ACCESS_AND_SHARE_MODES keyword. When an open request occurs, the first set of access and share modes are evaluated to see if they are valid for this request. If they are not, the next set of access and share modes are evaluated. This process continues until a set of access and share modes that are valid are found.

  In this example, the first open request will be granted read and write access to the file. The share modes are set to read and modify. Successive requests will not be allowed to use the first set of access and share modes so the second set, read and modify access modes and read and write share modes, will be granted.

```
  integer fitptr,                  ! Pointer to the file information table.
+   mode                           ! Access and share modes returned.

  call FILEIS(fitptr,              ! Create a file information table
+  '$LOCAL_FILE_NAME', 'a_file', !   for an indexed-sequential file.
+  '$KEY_LENGTH', 7,               ! Define key length.
+  '$MAXIMUM_RECORD_LENGTH', 132,! Define max record length.
+  '$MINIMUM_RECORD_LENGTH', 80, ! Define min record length.
+  '$ACCESS_AND_SHARE_MODES',     ! Define two sets of access, share modes.
+    '((READ,WRITE),(READ,MODIFY)),(READ,MODIFY),(READ,WRITE))')

  call OPENM(fitptr)               ! Open the file.  For the 1st request,
                                   !   access modes = read, write;
  call IFETCH(fitptr,              !   share modes = read, modify.
+   '$GLOBAL_ACCESS_MODE', mode) ! IFETCH returns an access mode of 3.
  call IFETCH(fitptr,              ! IFETCH returns a share mode of 7.
+   '$GLOBAL_SHARE_MODE', mode)
```

What Are Locks?

Keyed-file sharing is coordinated through the use of locks. A lock is a mechanism by which a task can restrict use of a keyed file or individual primary-key values in keyed files. The lock is owned by a particular instance of open for the file.

The part of the NOS/VE system software that manages locks is called the lock manager. In general, lock processing follows this pattern:

1. The lock manager receives a request for a lock on a nested file or record.
2. The lock manager determines whether the lock can be granted.
   a. If no conflicting lock exists, the lock manager grants the lock and notifies the requesting task.
   b. If a conflicting lock exists, the lock manager checks if the request specified waiting.
      1) If the request specified no waiting, the lock manager notifies the task requesting the lock that the record or file is currently locked.
      2) If the request specified waiting, the task is suspended until either:
         a) The lock is available (assuming no potential deadlock as described later under Lock Deadlock), or
         b) The timeout period elapses. The default timeout period is 60 seconds.

The lock manager also processes requests to clear locks and keeps track of locks that have expired (as described under Lock Expiration and Clearing).

NOTE

In general, when the Locks discussion describes two or more tasks requesting locks, the two or more tasks could actually be the same task with two or more instances of open of the same file. A lock belongs to a particular instance of open, and one task can request locks for more than one instance of open.

Reasons for Locks

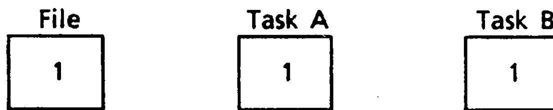
Locks are recommended for effective sharing of a keyed file. In fact, when more than one instance of open exists for a keyed file, NOS/VE requires that a task lock the record before it can replace or delete the record.

Locks ensures that:

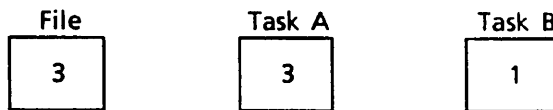
- Requests are processed in the sequence in which requests are issued.
- The operation is performed on the most up-to-date version.

To illustrate the need for locks, the following sequence of events describes two tasks using the same nested file without locks.

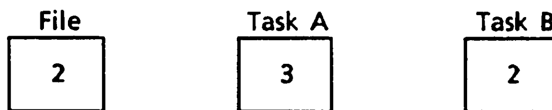
1. Two tasks both read the same record containing the value 1.



2. One task adds 2 to the value and replaces the record, containing the value 3, in the file.



3. The other task adds 1 to the value and replaces the record, containing the value 2, in the file.

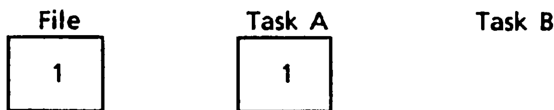


The work of one of the tasks has been overwritten.

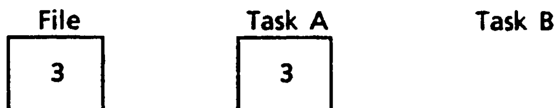
Next, consider the alternative in which locks are used.

1. A task locks and reads a record.

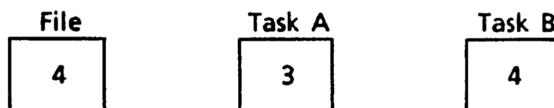
2. A second task attempts to lock and read the record but cannot because the record is already locked. It waits until the record is unlocked.



3. The first task adds 2 to the value, and replaces the record containing the value 3, in the file. It then unlocks the record.



4. The second task can now lock and read the record. It adds 1 to the value, and replaces the record, containing the value 4, in the file.



Lock Intents

Each lock has a lock intent. The lock intent indicates why the task is requesting the lock.

When more than one instance of open exists for a keyed file, only the owner of an Exclusive_Access or Preserve_Access_and_Content lock on the record (or the file) can replace or delete the record. However, the replace or delete operation does not take place until no unexpired Preserve_Content locks exist for the record.

Lock intents for file locks are described later under File Locks. The following lists describe the lock intents for record locks.

Exclusive_Access Lock Intent

Use the Exclusive_Access lock intent when:

- The task intends to issue write or delete requests for the locked primary-key value. The instance of open must have shorten or append access to the file.
- The task intends to deny all requests by other tasks to read, write, update, or delete the record or lock its primary-key value.

Preserve_Access_and_Content Lock Intent

Preserve_Access_and_Content lock intent is also known as write intent. Use this lock intent when:

- The task might issue write or delete requests for the locked primary-key value. Only one Preserve_Access_and_Content lock is allowed at a time for a key value.
- The task intends to allow positioning and read requests by other tasks, but denies their attempts to write, replace, or delete using the locked key value.
- The task intends to allow Preserve_Content lock requests by other tasks, but denies their requests for an Exclusive_Access or Preserve_Access_and_Content lock on the primary-key value.
- The owner of the Preserve_Access_and_Content lock can request a write, replace, or delete operation, but:
  - The write, replace, or delete operation does not begin until the conditions for an Exclusive_Access lock are met:
    - All read operations in progress for the record have completed.
    - All Preserve_Content locks for the record have expired or been cleared.
  - No read operations for the record can begin until the write, replace, or delete operation completes.

Preserve_Content Lock Intent

Preserve_Content lock intent is also known as read intent. Use the Preserve_Content lock intent when:

- The task does not intend to issue write, replace, or delete requests for the locked primary-key value.
- More than one instance of open exists, and the task intends to prevent all update attempts, including those of the lock owner. However, if the Preserve_Content lock owner is the only existing instance of open, the lock does not prevent updates.
- The task intends to allow positioning and read requests by other tasks, but denies their write, replace, and delete requests.
- The task intends to allow Preserve_Content and Preserve_Access_and_Content locks by other tasks, but denies their Exclusive_Access lock requests.

  Multiple Preserve_Content locks are allowed at a time, but only one Preserve_Access_and_Content lock. Thus, multiple tasks can be reading the record, but only one task can be waiting to write, replace, or delete the record.

Lock Renewal and Lock_Intent Changing

The owner of a lock can renew the lock by issuing a lock request without an intervening unlock request. The lock renewal restarts the expiration time for the lock.

The lock renewal can also change the lock_intent from Preserve_Access_and_Content to Exclusive_Access and vice versa.

An instance-of-open owning a Preserve_Content key lock or file lock cannot be granted an Exclusive_Access or Preserve_Access_and_Content file lock until it unlocks its Preserve_Content lock.

Depending on the lock_intents, a request for a lock that you already hold may result in an error. To see the possible outcomes, see Lock Conflict Tables at the end of this locking discussion.

File Locks

Your program should request a file lock when it needs locks on many key values at the same time. A file lock is a lock on all primary-key values for a nested file.

In general, the rules for using file locks are the same as those for locks on individual primary-key values.

The effect of the lock intent of a file lock is as follows:

- Exclusive_Access

  Used when the nested file is to be updated.

  Allows access to records in the nested file only by the instance of open holding the file lock; all requests by other instances of open are denied including all lock requests.

- Preserve_Access_and_Content

  Used when the instance of open intends to read records in the nested file and may update records later. It allows the holder to do updates, but prevents all other instances of open from updating.

  Allows all instances of open to read the file and allows Preserve_Content locks for records in the file or the file as a whole, but denies all Exclusive_Access and Preserve_Access_and_Content locks (except a file lock for the nested file by the same instance of open).

- Preserve_Content

  Used to prevent file updates if the file is shared. (The lock owner can update the file if no other instance of open exists.)

  Allows any number of Preserve_Content locks and one Preserve_Access_and_Content lock for each primary-key value and for the file as a whole, but denies all Exclusive_Access lock requests.

For further details on the file and key-value locks that can co-exist, see Lock Conflict Tables.

A file lock is required when your program needs more than 1024 locks at a time because 1024 is the maximum number of locks allowed for an instance of open. An attempt to exceed this limit returns the nonfatal $ERROR_STATUS value AA2115.

The number of locks allowed also depends on the FILE_LIMIT attribute value. The lock manager tracks all locks for a file in another file called the lock file. The lock file size cannot exceed 90% of the FILE_LIMIT value and, if an operation would cause the lock file to be more than 50% full, the operation is not allowed to begin and the fatal $ERROR_STATUS value AA6010 is returned.

Waiting for a Lock

When a conflicting lock exists, but no deadlock, a call requesting a lock waits for the lock if the $WAIT_FOR_LOCK value in the FIT is TRUE.

A lock request waits until the lock is available or the lock timeout period has passed. If the lock request times out, the call returns the $ERROR_STATUS value AA2055.

The default timeout period is 60 seconds. However, each task can specify how long it waits for a lock by creating and initializing an NOS/VE integer variable named AAV$RESOLVE_TIME_LIMIT. The value assigned to the variable is the new lock timeout period in seconds. Do not set the lock timeout period so that it is longer than the LOCK_EXPIRATION_TIME attribute value. The default LOCK_EXPIRATION_TIME is 60 seconds.

For example, the following call executes the NOS/VE statement VAR/VAREND to change the timeout period to 45 seconds:

```
call sclcmd ('var AAV$RESOLVE_TIME_LIMIT: integer=45; varend')
```

Lock Expiration and Clearing

An expired lock and a cleared lock are not the same:

- A cleared lock no longer exists; the lock manager has discarded it.
- An expired lock is no longer effective in preventing access by other tasks. However, an expired lock prevents file access by its owner (except IFETCH and STOREF calls and an UNLOCKF or UNLOCKK call that clears the expired lock). This is done so that the owner of the lock is notified of its expiration.

A lock is cleared when one of these events occurs:

- The task with the lock issues an unlock request for the lock.
- The task closes the instance of open to which the lock applies.
- The request for the record lock specified automatic unlock, and the task issues any request for the instance of open (other than an IFETCH or STOREF call).

  In general, the automatic unlock occurs when the request is issued. The exception is for an update request for the locked record for which the lock is kept until the update operation completes.

  For example, if a task issues a lock on record 1 and then issues a request to replace record 1, the lock manager automatically clears the lock on record 1 after the replace operation. Similarly, if a task issues a lock on record 1 and then issues a request to position the file at record 2, the lock manager automatically clears the lock on record 1, before positioning the file at record 2.

The expiration of a lock granted during a parcel aborts the parcel. It aborts the parcel for the instance of open to which the parcel applies. For more information on lock expiration during a parcel, see chapter 4, Parcels.

How a Lock Expires

A lock expires when the following sequence of events occurs:

1. Its expiration time has passed since the lock was granted.
2. Another task issues a request specifying waiting that would be denied if the lock was effective. (The request is granted.)

The number of milliseconds in the lock expiration time is specified by the file attribute, LOCK_EXPIRATION_TIME. The default value is 60,000 milliseconds (60 seconds). To set an unlimited expiration time so that locks do not expire, set the attribute value to 0.

An expired lock is no longer effective in preventing access to the file or record by other tasks. However, it does prevent the task holding the expired lock from accessing records in the file.

The task holding the expired lock is prevented from accessing any record in the file until it clears the expired lock. This notifies the task that a lock has expired.

For example, consider the following sequence of events:

1. Task 1 requests a Preserve_Access_and_Content lock on record 1 in nested file 1 without automatic unlock. The lock is granted.

2. The expiration time passes.

3. Task 1 reads record 1 from nested file 1. The read request restarts the expiration time count.

   (The lock has not yet expired because no other task has issued a request for the record that a Preserve_Access_and_Content lock should prevent. The lock is not unlocked because automatic unlock was not requested for the lock.)

4. The expiration time passes again.

5. Task 2 requests a Preserve_Content lock on record 1 in nested file 1. (The Task 1 lock does not expire because a Preserve_Access_and_Content lock does not prevent Preserve_Content locks.)

6. Task 3 requests, with waiting, a Preserve_Access_and_Content lock on record 1 in nested file 1. (The Task 1 lock expires because a Preserve_Access_and_Content lock should prevent additional Preserve_Access_and_Content locks.)

7. Task 1 attempts to read record 2 in nested file 1, but instead the request terminates with a nonfatal error, notifying Task 1 that it has an expired lock. Task 1 must clear the expired lock before it can successfully request any record in nested file 1.

A task is notified of lock expiration for the currently selected nested file only. The expiration of locks in a previously selected nested file does not affect the task unless it re-selects the nested file and attempts a file operation.

Expired Lock Conditions

These are the nonfatal $ERROR_STATUS values returned for an expired lock:

AA2790

The sequential read (get_next) failed due to an expired lock.

AA2805

The primary-key value or file could not be locked due to an expired lock.

Lock Deadlock

A deadlock is a situation in which two or more tasks need a lock already held by another task in the group of tasks. For example, the following situation is a deadlock:

- Task 1 has a lock on record 1 and needs a lock on record 2.
- Task 2 has a lock on record 2 and needs a lock on record 3.
- Task 3 has a lock on record 3 and needs a lock on record 1.

If none of the tasks releases the lock it holds, none of the tasks can complete.

A deadlock can occur either when tasks are waiting for a lock or when tasks are each repeatedly requesting a lock. The lock manager can detect the deadlock when the tasks are actually waiting for a lock; it cannot detect a deadlock when tasks are repeatedly requesting locks.

When the lock manager receives a lock request indicating that the task wants to wait until the lock is available, it checks for a possible deadlock. To do so, it checks whether other tasks are waiting for locks held by the requesting task. If it detects a potential deadlock, it terminates the request with a nonfatal error.

If the deadlock is with another task, it returns error AA2040. If the deadlock is a self-deadlock (the requesting task already has the requested lock), it returns error AA2045.

To prevent a deadlock that the lock manager cannot detect, a task should limit the number of times it repeatedly requests a lock without waiting. After a fixed number of attempts, it should do one of the following:

- Issue a lock request with waiting in which case the lock manager can notify it that a potential deadlock exists.
- Assume that a potential deadlock exists and clear the locks it holds.

Lock Conflict Tables

The outcome of a request for a lock that has already been granted depends on:

- The lock intents of the existing and requested locks.
- Whether the request is from the same instance of open holding the lock.
- Whether the conflicting locks are key-value locks or file locks.

Tables 3-1 and 3-2 give the outcomes when the requested and existing locks are either both key-value locks or both file locks.

Table 3-1. When the Lock Request is From the Same Instance of Open

Existing Lock Intent	Requesting Preserve_Content Lock Intent	Requesting Preserve_Access_ and Content Lock Intent	Requesting Exclusive_Access Lock Intent
Preserve_ Content	Renews	Rejects	Rejects
Preserve_ Access_and_ Content	Rejects	Renews	Renews
Exclusive_ Access	Rejects	Renews	Renews

Table 3-2. When the Lock Request is From Another Instance of Open

Existing Lock Intent	Requesting Preserve_Content Lock Intent	Requesting Preserve_Access_ and Content Lock Intent	Requesting Exclusive_Access Lock Intent
Preserve_ Content	Grants	Grants	Depends
Preserve_ Access_and_ Content	Grants	Depends	Depends
Exclusive_ Access	Depends	Depends	Depends

Definition of Results

Grants — Grants the lock.

Renews — Renews the lock, restarting its lock expiration time and changing the lock intent if requested.

Rejects — Rejects the request and returns nonfatal status AA2080.

Depends — The result depends on whether waiting is requested:

- No waiting requested: Rejects request and returns nonfatal status AA2075
- Waiting requested: Grants the lock unless:
  - Opens belong to the same task: Rejects request and returns nonfatal status AA2045
  - Opens belong to different tasks: Grants the lock unless:
    - Deadlock detected and returns nonfatal status AA2040
    - Timeout period elapses and returns nonfatal status AA2055

Tables 3-3 and 3-4 give the outcomes when the existing and requested locks are not the same kind of lock (file locks or key-value locks).

Table 3-3. When the Lock Request is From the Same Instance of Open

Existing Lock Intent	Requesting Preserve_Content Lock Intent	Requesting Preserve_Access_ and Content Lock Intent	Requesting Exclusive_Access Lock Intent
Preserve_ Content	Grants	Grants	Self-Deadlock
Preserve_ Access_and_ Content	Grants	Self-Deadlock	Self-Deadlock
Exclusive_ Access	Self-Deadlock	Self-Deadlock	Self-Deadlock

Table 3-4. When the Lock Request is From Another Instance of Open

Existing Lock Intent	Requesting Preserve_Content Lock Intent	Requesting Preserve_Access_ and Content Lock Intent	Requesting Exclusive_Access Lock Intent
Preserve_ Content	Grants	Grants	Depends
Preserve_ Access_and_ Content	Grants	Depends	Depends
Exclusive_ Access	Depends	Depends	Depends

Definition of Results

Grants	Grants the lock.
Renews	Renews the lock, restarting its lock expiration time and changing the lock intent if requested.
Self-Deadlock	Rejects the request and returns nonfatal status AA2045.
Depends	The result depends on whether waiting is requested:

- No waiting requested: Rejects request and returns nonfatal status AA2075
- Waiting requested: Grants the lock unless:
  - Opens belong to the same task: Rejects request and returns nonfatal status AA2045
  - Opens belong to different tasks: Grants the lock unless:
    - Deadlock detected and returns nonfatal status AA2040
    - Timeout period elapses and returns nonfatal status AA2055

Parcels 4

File-Spanning Parcels 4-1

How to Use Parcels in FORTRAN Programs 4-2
Required Attribute 4-2
Parcel Processing Outline 4-2
Record Access During a Parcel 4-4
FIT Values Affecting Parcels 4-5
Lock Expiration During a Parcel 4-6
Using the Parcel Log 4-7
Calls Dissallowed During a Parcel 4-9

File-Level Parcels 4-10

Parcel Program Example 4-11

Parcels 4

This chapter describes parcels and how to use them in FORTRAN programs.

Two types of parcels are available: a file-spanning parcel and a file-level parcel that applies only to a single keyed file. The first section describes the file-spanning parcel; the differences between file-spanning parcels and file-level parcels are described later in this chapter under File-Level Parcels.

File-Spanning Parcels

A file-spanning parcel is a series of update operations that form a logical group. The update operations can apply to more than one instance of open and to more than one keyed file, but the operations must all be performed by the same task.

After a program signals the beginning of a parcel and specifies the instances of open to which the parcel applies, all update operations to those instances of open belong to the parcel. The parcel ends when the program chooses to commit or abort the parcel. Committing a parcel causes its update operations to become permanent. Aborting a parcel discards the updates in the parcel, leaving the keyed files as they were before the parcel began.

Using parcels requires some system overhead because the system must record each parcel update operation in temporary system space so that if the parcel is aborted, all records in the update operation are restored.

Using parcels offers advantages when logical update operations affect more than one record. For example, parcel use enables:

- Easy implementation of an interactive veto option to undo a set of update requests.
- File recovery such that the damaged file is recovered to a point before or after a logical update, not before or after an individual update request. (The file recovery requires that an update recovery log be maintained for the file. See Protecting Your Keyed Files in the NOS/VE Advanced File Management manual.)

How to Use Parcels in FORTRAN Programs

You can use parcels in FORTRAN programs with keyed-file interface calls. See chapter 6, Keyed-File Interface Calls, for a description of the calls referred to in this chapter. This section describes how to use parcels and the calls you need to perform parcel processing.

Required Attribute

The system allows use of parcels to update a keyed file only if the logging_options attribute for the file includes the option enable_parcels. The logging_options attribute value is set when the keyed file is first created. It can be changed later using the NOS/VE command CHANGE_FILE_ATTRIBUTES.

NOTE

While the logging_options attribute of a file includes enable_parcels, each instance of open must request modify access if it allows write concurrency. In other words, if the open request allows no sharing or read-only sharing, it does not need to specify modify access; otherwise, it must specify modify access.

Parcel Processing Outline

Each task can have only one file-spanning parcel in progress at a time. The following is an outline of the steps involved in processing a parcel.

1. The program should check that the file (or files) to which the parcel is to apply is a keyed file and has parcels enabled. A file is a keyed file when its file_organization attribute is indexed_sequential or direct_access. Parcels are enabled when the file's logging_options attribute includes enable_parcels.

2. The task opens the file (or files) to which the parcel is to apply.

3. To signal the beginning of the parcel, the task calls PBEGIN.

   The parcel applies to all keyed files that are open and have parcels enabled. Or, if desired, the call can limit the parcel to a set of keyed files. (A file-level parcel is created for each file.)

4. The task issues the update requests that belong to the parcel. These can include put, replace, and delete requests. Each update request is processed as follows:

   a. The system requests an Exclusive_Access lock for the specified primary-key value. The lock prevents all other instances of open from reading or updating the record until the parcel completes. This includes attempts to read the record by an alternate key.

   b. The update request is processed as appropriate for a put, replace, or delete as follows:

      - For a put request, the system saves the key of the record to be added, puts the primary-key value in the primary index, and puts its alternate-key values in the alternate indexes.

- For a delete request, the system saves the record to be deleted. It then deletes the record data and the primary-key value from the file. However, any alternate-key values remain in the alternate indexes until the parcel is committed.

- For a replace request, the system saves the record to be replaced. It then replaces the record data in the file. Any new alternate-key values are added immediately to the alternate indexes, but the existing alternate-key values remain in the alternate indexes until the parcel is committed.

5. The task can fetch the status of any parcel at any time by calling PDETERM. The call also returns the message written by the last parcel call that included a message for the parcel, if any. (Each parcel call can store a message in the parcel log.)

6. The parcel is committed or aborted.

   a. A parcel is aborted by a PABORT call issued by the system or by the task that began the parcel.

   NOTE

   The system aborts a file-level parcel if the instance of open to which the parcel applies is closed before the parcel is committed. A subsequent attempt to commit the file-spanning parcel will abort any other file-level parcels.

   The abort_parcel call performs the following restoration:

   - Deletes records put by the parcel, restores records replaced by the parcel, and restores deleted records to the primary index.
   - Restores the nested-file selection and alternate-key selection to those in effect when the parcel began.
   - Repositions the file to its position when the parcel began.
   - Writes an abort record for the parcel to the parcel log. It also stores the message, if any, specified on the call in the log.

   b. The task that began the parcel can commit the parcel by calling PCOMMIT. A successful commit performs the following:

   - Removes the alternate-key values of deleted and replaced records from the alternate indexes.
   - Records the parcel updates in the log if an update recovery log is maintained for the keyed file. (The parcel is recorded as a unit so that a recovery restores the parcel as a unit.)
   - Writes a commit record for the parcel to the parcel log. It also stores the message, if any, specified on the call in the log.

   If the commit fails, the system transforms it to an abort and performs the abort processing listed under step 6a.

7. After completing the commit or abort processing, the system releases *all* locks set for the instances of open that were included in the parcel.

Record Access During a Parcel

The locks granted in a parcel determine the access that each instance of open has to the records locked during the parcel. All locks granted during a parcel remain in effect until the end of the parcel.

Because the lock granted to parcel update requests is for Exclusive_Access and the lock is granted to only one instance of open, only that instance of open can read or update the record. All other instances of open are prevented from reading or updating the locked record during the parcel.

For example, suppose, as part of a parcel, an instance of open replaces a record with alternate-key value XYZ:

- The instance of open holding the lock can read all records with alternate-key value XYZ.
- Any other instance of open cannot read the replaced record. It can read other records having alternate-key value XYZ, but an attempt to read the replaced record returns a nonfatal status AAE$KEY_FOUND_LOCK_NO_WAIT indicating that the record is locked.

Locks do not prevent calls that rewind the file or skip records. Therefore, during the parcel, any instance of open can issue those calls. However, the effects of the update calls in the parcel could affect a file positioning call so that it is ineffective.

A Preserve_Content lock granted before or during a parcel prevents updating of its record during the parcel. This is because a Preserve_Content lock does not allow updating of the record, and the Preserve_Content lock cannot be released during the parcel. (It is released at the end of the parcel with all other locks.)

While the record is being updated by a parcel, other tasks cannot read the record because its primary-key value is locked. Other tasks also cannot write a record whose alternate-key value is the same as that of a record added or deleted by the parcel. (This applies only to alternate keys that do not allow duplicate values.)

For example, assuming the alternate key does not allow duplicate values, if a record with alternate-key value XYZ is deleted by a parcel, another task cannot write a new record with alternate-key value XYZ to the file by another instance of open until the parcel is committed.

FIT Values Affecting Parcels

In general, the FORTRAN parcel calls, themselves, do not use FIT values as defaults or store information in the FIT. However, the effects of the following FIT values should be noted:

$LOGGING_OPTIONS

To be able to use parcels for a file, its logging_options attribute must include enable_parcels. Thus, the call that creates the FIT should store the value 'EP' as the $LOGGING_OPTIONS FIT value so that the attribute is validated when the file is opened.

While the logging_options attribute of a file includes enable_parcels, each instance of open must request modify access if it allows write concurrency. In other words, if the open request allows no sharing set or read-only sharing, it does not need to specify modify access, otherwise; it must specify modify access.

$AUTOMATIC_UNLOCK

This value is ignored by calls inside a parcel. Inside a parcel, no lock can be released. All locks granted inside a parcel are released when the parcel ends.

$LOCK_INTENT

Because locks cannot be released inside a parcel, a Preserve_Content lock prevents updates to its record during the parcel. This is because a Preserve_Content lock prevents updating of the record and locks cannot be released inside a parcel.

Lock Expiration During a Parcel

A file-spanning parcel is a collection of file-level parcels. The expiration of a lock granted in one keyed file aborts that file-level parcel. The task is notified if it attempts any update on that keyed file. Access to other keyed files included in the file-spanning parcel continues normally until a file-spanning parcel commit is attempted. Then the file-spanning parcel is aborted.

NOTE

To ensure that locks will not expire during a parcel, either open the file for exclusive access (no sharing), allow one instance of open for the file, or ensure that the LOCK_EXPIRATION_TIME attribute for the file is 0. For more information about lock expiration, see Lock Expiration and Clearing in chapter 3.

If one of the file-level parcels is aborted, the task is notified when it issues a request specifying the instance of open to which the file-level parcel applies. The request returns a status describing why the parcel was aborted.

To continue using the instances of open for which a parcel was aborted, the task must:

1. Check for the parcel_abort status after each request in the parcel, and
2. For a file-level parcel:

   When a request returns the parcel_abort status, either call PABORT for the parcel or CLOSEM for the instance of open to which the expired lock belonged. This notifies the system that the parcel_abort status was received.

   (The call issued in response to the parcel_abort status does not abort the parcel; the purpose of the call is to acknowledge the lock expiration so you can continue; the lock expiration has already aborted the parcel.)

   For a file-spanning parcel:

   Call PABORT to abort each of the file-level parcels in the file-spanning parcel.

Using the Parcel Log

The system stores information about a parcel in a log. The PBEGIN call specifies the log to be used for the parcel. The default is the log specified by the SCL variable AAV$LOG_SELECTION, which is initially set by the system prolog, or the default system log, $SYSTEM.AAM.SHARED_RECOVERY_LOG. (If both logs exist, the log specified by the SCL variable AAV$LOG_SELECTION is used.) However, you can specify another log. The log must have been created by the Administer_Recovery_Log utility (described in the NOS/VE Advanced File Management manual).

Each parcel call (begin, abort, and commit) writes a record to the parcel log. It identifies the parcel and the action taken (begin, abort, or commit). The call can also store an optional message in the log.

The message stored in a parcel log is a string of data, up to the maximum length of a keyed-file record. A FORTRAN parcel call specifies the message by the name of the character variable containing the data.

A task can call PDETERM to determine the state of a file-spanning parcel and fetch the most recent message, if any, stored in the parcel log record. The task can call PDETERM at any time to fetch information about any file-spanning parcel for which it has the user parcel name, the catalog path of the parcel log, and the file-spanning parcel name. The task that calls PDETERM does not have to be the one that called PBEGIN.

For Better Performance

The log to be used for the parcel can be the same as the recovery log; however it is more efficient to specify a separate log for a parcel whenever possible. This creates a smaller log for the PDETERM call to search through to retrieve the status and the messages.

PDETERM returns the parcel state as one of the following:

Parcel State	Parcel State Name	Description
0	Parcel active.	The call found a begin record for the parcel on the log, but no commit or abort record. The call returns the message it finds in the begin record, if any.
1	Parcel committed.	The call found a commit record for the parcel on the log. The call returns the message it finds in the commit record, if any. If no message is found, the call returns the message, if any, from the begin record.
2	Parcel aborted by system.	The call found an abort record for the parcel on the log. The call returns the message it finds in the abort record, if any. If no message is found, the call returns the message, if any, from the begin record.
3	Parcel aborted by user.	The call found an abort record for the parcel on the log. The call returns the message it finds in the abort record, if any.
4	Parcel not found.	The call found no records for the specified parcel in the log. Check to make sure that the correct log is specified.
5	Parcel indeterminate.	The call may have found a begin record for the parcel, but also found indication of a catastrophic, unrecoverable error that prevented completion of the parcel. Check to make sure that a log exists or that the task has access to the log.

Calls Dissallowed During a Parcel

During a parcel, the following calls are not allowed for the instances of open to which the parcel applies; the calls are allowed for any other instances of open.

Call	Reason
LOCKF	A lock intent of 'PAC' or 'PC' cannot be granted during a parcel.
UNLOCKF	The program can explicitly request locks during a parcel, but it cannot explicitly unlock locks. (All locks granted before or during the parcel are unlocked at the end of the parcel.)
PBEGIN	Each task can have only one parcel in progress at a time so PBEGIN is disallowed inside a parcel.
CLOSEM	Closing an instance of open to which a file-level parcel applies aborts that file-level parcel.
RMKDEF	The alternate-key selection can change during a parcel, but the alternate-key definitions must not change.
KEYLIST KLCOUNT	The alternate-key values in a key list cannot be counted or returned during a parcel.
RSBUILD RSCOMB RSPUT RSDLTE	Result sets cannot be created during a parcel, but result sets can be used to read records during a parcel.

File-Level Parcels

The previous sections describe the use of file-spanning parcels. A second, more restricted, type of parcel is also available, the file-level parcel.

A file-level parcel is like a file-spanning parcel with the following restrictions:

- The parcel can apply to only one instance of open.
- The parcel log cannot be selected or accessed.
- You cannot store messages for a parcel or fetch the parcel state.

For Better Performance

If the parcel is to apply to only one instance of open, use a file-level parcel, not a file-spanning parcel. A file-spanning parcel requires more system overhead.

Each instance of open is allowed to have only one file-level parcel in progress.

File-level parcels use the following calls:

PABORT

PBEGIN

PCOMMIT

For a description of these calls, and the parameter requirements for keyed-file parcels, see chapter 6, Keyed-File Interface Calls.

Parcel Program Example

The following FORTRAN program uses a parcel to write a record to a master file and its shadow file; the default parcel log is used. A parcel abort causes the parcel to be restarted; the program attempts the parcel no more than five times.

```
      PROGRAM parcel

      INTEGER
     + fit1,             ! Pointer to the File Information Table
     + fit2,             ! Pointer to the File Information Table
     + fitlist(2),       ! Array of FIT pointers for processing parcels
     + attempt,          ! Number of attempts for the parcel
     + condition_code,   ! Condition code returned as error status
     + length            ! Length of condition_name returned from CONDSYM

      CHARACTER*31
     + prclnam          ! Parcel name

      CHARACTER*8
     + wsa,             ! Working storage area
     + condition_name   ! Condition name returned by CONDSYM, deciphered
                        ! From the condition_code

C      Create FIT for existing file MASTER_FILE and check for required
C      logging option.

      CALL fileis (fit1,
     +           '$local_file_name', 'MASTER_FILE',
     +           '$key_length', 3,
     +           '$maximum_record_length', 8,
     +           '$minimum_record_length', 8,
     +           '$logging_options', 'enable_parcels')

C      Create FIT for existing file SHADOW_FILE and check for required
C      logging option.

      CALL fileis (fit2,
     +           '$local_file_name', 'SHADOW_FILE',
     +           '$key_length', 3,
     +           '$maximum_record_length', 8,
     +           '$minimum_record_length', 8,
     +           '$logging_options', 'enable_parcels')

C     Create the files (NEW) and position them at rewind (R).

      CALL openm (fit1, 'NEW', 'R')
      CALL openm (fit2, 'NEW', 'R')
```

```
C     Fetch the error status of OPENM and decipher it with the FORTRAN
C     error-processing routine CONDSYM.

      CALL ifetch (fit1, '$error_status', condition_code)
      CALL condsym (condition_code, condition_name, length)

      IF (condition_code .NE. 0) THEN
        print *,'Error encountered. Condition name is ', condition_name
        STOP
      END IF

      CALL ifetch (fit2, '$error_status', condition_code)
      CALL condsym (condition_code, condition_name, length)

      IF (condition_code .NE. 0) THEN
        print *,'Error encountered. Condition name is ', condition_name
        STOP
      END IF

C      Create the list of FITs to which the parcel applies.

      fitlist(1) = fit1
      fitlist(2) = fit2

C      Assign data to the working storage area so the record can be
C      written to the file.

      wsa = 'KEY/DATA'

C      Attempt the parcel no more than 5 times.

      DO 20 attempt = 1, 5

        CALL pbegin ('my_parcel', fitlist, 2, condition_code, prclnam)
        CALL condsym (condition_code, condition_name, length)

        IF (condition_code .NE. 0) THEN
          print *, 'Error encountered.  Condition name is ',
     +              condition_name
          STOP
        END IF
```

```
C      Write record.

        CALL put (fit1, wsa, 8, 0, 0, 0, 0)

        CALL ifetch (fit1, '$error_status', condition_code)
        CALL condsym (condition_code, condition_name, length)

        IF (condition_code .NE. 0) THEN
          IF (condition_name .EQ. 'AA 2075') THEN
            CALL pabort (prclnam, condition_code) pabort (prclnam)
            IF (condition_code .NE. 0) THEN
              CALL condsym (condition_code, condition_name, length)
              print *,'Error encountered.  Condition name is ',
     +                condition_name
              STOP
            END IF
            GO TO 20    ! Go to the end of the DO Loop and retry parcel.
          ELSE
            print *,'Error encountered. Condition name is ',
     +               condition_name
            STOP
          END IF
        END IF

        CALL put (fit2, wsa, 8, 0, 0, 0, 0)

        CALL ifetch (fit2, '$error_status', condition_code)
        CALL condsym (condition_code, condition_name, length)

        IF (condition_code .NE. 0) THEN
          IF (condition_name .EQ. 'AA 2075') THEN
            CALL pabort( prclnam, condition_code)
            IF (condition_code .NE. 0) THEN
              CALL condsym (condition_code, condition_name, length)
              print *,'Error encountered.  Condition name is ',
     +                condition_name
              STOP
            END IF
            GO TO 20    ! Go to the end of the DO Loop and retry parcel
          ELSE
            print *,'Error encountered. Condition name is ',
     +               condition_name
            STOP
          END IF
        END IF
```

```
C      If put was successful, commit parcel and quit.

        CALL pcommit (prclnam, condition_code)
        IF (condition_code .EQ. 0) THEN
          CALL closem (fit1, 'U')
          CALL closem (fit2, 'U')
          print *, 'Parcel committed. '
          print *, 'Program ended with a normal condition code.'
          STOP
        END IF

20    CONTINUE          ! End of DO loop.

      CALL closem(fit1, 'U')
      CALL closem(fit2, 'U')
      STOP
      END
```

Result Sets 5

What Are Result Sets? 5-1

How to Use Result Sets in FORTRAN Programs 5-2
Result Set Validity 5-3
Keeping the Result Set Accurate 5-3
Keeping the Result Set Accurate Within a Single Instance of Open 5-3
Keeping the Result Set Accurate Within a Single Job 5-3
Keeping the Result Set Accurate Across Jobs 5-3
Recovering from Result Set Read Errors 5-4
Result Set Files 5-4
Combining Result Sets 5-4
Combination Operations 5-5
Placement of the Combined Result Set 5-5
Adding Or Deleting Key Values 5-6

Result Sets 5

What Are Result Sets?

A result set is a set of primary-key values. It provides a means of reading a logical set of records from a keyed file.

A result set begins as a list of primary-key values, retrieved using a key-value range for the currently selected key. The range is specified in the same way as the range on an KLCOUNT call. However, unlike a simple key list, a result set can be combined and modified.

A result set can be combined with other result sets using the logical operations AND, OR, and XOR. It can be modified by adding and deleting values from the result set. Also, the result set can be used to read either the set of records referenced in the result set or (for indexed-sequential files) all records not referenced in the result set.

How to Use Result Sets in FORTRAN Programs

The following is a general outline of the steps by which a FORTRAN program creates and uses a result set to read a set of records:

1. Open the keyed file by calling OPENM. If the result set is for a nested file other than the default nested file ($MAIN_FILE), specify the nested file by storing the nested file as the FIT value for $NESTED_FILE_NAME.

2. Open the result set file by calling RSOPEN. RSOPEN returns the result_set_id, which is used by subsequent calls to reference the open result set.

3. If the existing result set in the result set file is to be discarded, clear the result set by calling RSCLEAR.

4. Change the result set by calling:

   RSBUILD — Gets the set of primary-key values specified on the call and combines the new set with an existing result set.

   (If you want to use an alternate key, it must be stored as the $KEY_NAME FIT value before the RSBUILD call. You must use alternate keys for a direct-access file. A range of primary-key values cannot be specified for a direct-access file because the primary-key values are not ordered in a direct-access file.)

   RSCOMB — Combines two existing result sets.

   RSPUT — Adds a primary-key value to a result set.

   RSDLTE — Deletes a primary-key value from a result set.

5. If an alternate key is currently selected, select the primary key by calling STOREF to store the value $PRIMARY_KEY as the $KEY_NAME FIT value.

6. Read records from the keyed file by calling RSGETN. RSGETN allows you to do either of these actions:

   - Read the records that are in the result set.
   - Read the records that are not in the result set (indexed-sequential files only).

7. Fetch information about the result set at any time while the result set is open by calling RSINFO.

8. Reposition the result set (if appropriate) using the following calls:

   RSREWND — Position the result set at its beginning.

   RSSTART — Position the result set at the specified record.

   RSSKIP — Position the result set forward or backward a specified number of key values. This can be done only after the result set position has been established by a get, rewind, or start result set call.

9. Close the result set by calling RSCLOSE.

10. Close the keyed file by calling CLOSEM.

Result Set Validity

You can use a result set only for the file for which it was built. This restriction exists because the result set stores the global file name and the currently selected nested file when the result set is first opened.

A result set cannot be used with a copy of the original data file or another cycle of the file or another nested file in the data file. For example, if a result set is built for a temporary file, the result set cannot be used to read a permanent copy of the file. This is because the permanent copy has a different global file name.

Result sets to be combined must apply to the same nested file and file cycle. However, more than one key for the nested file can be used to build a result set. For example, the result set could be started while the primary key is selected and then added to after selection of an alternate key.

The accuracy of a result set is ensured only until the nested file is updated. At that time, key values of records referenced by the result set could be changed. When writing a program that uses result sets, you must determine whether the result set must be accurate when it is used to read records. If accuracy is not required, updates to the data file can continue while result sets are built and used.

Keeping the Result Set Accurate

If a program using a result set requires that the result set be accurate, it must ensure that the data file is not updated from the time you start using the result set until you are finished with the result set. How this is done depends on whether the result set is created and used within a single instance of open, within a single job, or across jobs.

Keeping the Result Set Accurate Within a Single Instance of Open

When the result set is created and used within a single instance of open, updates can be prevented by calling LOCKF before beginning to create the result set. The LOCKF call should request a Preserve_Content file lock to allow the nested file to be read, but not updated. The lock should be held until all use of the result set has been completed.

Keeping the Result Set Accurate Within a Single Job

When the result set is created and used within a single job, data file updates can be prevented by attaching the data file so that the specified access modes and share modes do not include append or shorten access modes. This prevents updating of the file while it is attached to the job.

Keeping the Result Set Accurate Across Jobs

When the result set is to be created and used across jobs, data file updates can be prevented by creating a permit for the data file that applies to all users (a public permit) that omits the append and shorten permissions. Also, to be used across jobs, the result set file must be a file in a permanent file catalog.

Recovering from Result Set Read Errors

If the file could be updated between the time the result set is built and the time it is used, the program should check for possible errors returned by RSGETN calls. Calls to read a record could fail because a primary-key value is locked or because the record for the primary-key value has been deleted. Because the errors are nonfatal, they do not terminate the program so the sequence of reads can continue.

To recover from a lock conflict while the file is shared, the program could retry reading the record. The retry method used depends on the result_set_not parameter value. The result_set_not parameter determines whether the call gets a record that is referenced in the result set or a record that is not referenced in the result set.

To retry a get call (result_set_not is 'NO'), call RSSKIP to move the result set back one value and then retry the read. Or, the program could call RSINFO to get the previous_key value. The program could then call GET using the previous_key value. With $GET_AND_LOCK and $WAIT_FOR_LOCK set, the GET call waits for the record until it can read it.

To retry a get_not call (result_set_not is 'YES'), no repositioning is necessary. The program can retry the read by calling RSGETN again.

Result Set Files

Result sets reside in sequential files, called result set files. The RSOPEN call specifies the result set file. If the specified file does not exist, RSOPEN creates the file.

The first RSOPEN call for a result set stores file attribute values that identify the file as a result set file. If the result set exists already, the RSOPEN call checks that the specified file is a result set file.

NOTE

To preserve the integrity of the result set, do not perform any operations except result set operations on result set files.

Combining Result Sets

The RSBUILD and RSCOMB calls can combine result sets.

- An RSCOMB call combines two existing result sets.
- An RSBUILD call combines an existing result set (called its source result set) with a new result set.

Combination Operations

Result sets can be combined by one of these three operations as specified by the logical_operation parameter on the call. The parameter can specify one of these integer values:

0 (AND)

The combined result set is the intersection of the result sets. It contains only those key values that belong to both of the sets.

1 (OR)

The combined result set is the union of the result sets. It contains all key values from both result sets.

2 (XOR)

The combined result set is the union of the result sets without the intersection of the result sets. It contains all key values from each of the result sets that do not belong also to the other result set.

Placement of the Combined Result Set

On the RSBUILD and RSCOMB calls, you specify the source result set as an existing result set. The combined result set can overwrite the source result set or be written to another result set file called the target result set.

The placement of the combined result set is determined by the value of the new_result_placement parameter on the call. The parameter can specify one of these integer values:

0 (Result in Source)

The combined result set overwrites the source result set. For RSCOMB, the result set overwritten is always the second of the source result sets specified. Use this value only when the source result set is no longer needed. It can also be used by an RSBUILD call when the source and target result sets are the same. (The source and target result sets cannot be the same for an RSCOMB call.)

1 (Result in Target)

The combined result set is written to the target result set. Use this value when the source_result_set is to be saved for later use. It is also used on the initial RSBUILD call for a new result set.

2 (Result in Fastest Place)

The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the variable specified by the actual_result_set placement parameter on the call. Use this value when the source result set is no longer needed and the source and target result sets differ.

Adding Or Deleting Key Values

The RSPUT and RSDLTE calls add or delete a primary-key value in the result set. These calls are for specifying a single primary-key value, instead of the range of values specified by an RSBUILD call.

For Better Performance

In cases where several scattered primary-key values are to be added or deleted in a result set and the result set is large, calls to directly add or delete individual values are not the most efficient method of producing the target result set.

It is more efficient to form a temporary result set containing the individual primary-key values and combine the temporary result set with the source result set to form the target result set.

If possible, put the primary-key values into the result set in ascending order. This builds the result set more efficiently.

To add several individual primary-key values to a large result set:

1. Call RSPUT to put each primary-key value to be added into a temporary result set.

2. Combine the result sets using an OR (1) operation.

To delete several individual primary-key values from a large result set:

1. Call RSPUT to put each primary-key value to be deleted into a temporary result set.

2. Call RSCOMB specifying the original result set as the first_source_result_set and the temporary result set as the second_source_result_set. Combine the result sets using an XOR (2) operation; specify result_in_source (0) to overwrite the temporary result set.

3. Combine the temporary result set created by step 2 with the original result set using an AND (0) operation. (This step is required only when a record to be deleted may not have been in the original result set.)

Keyed-File Interface Calls 6

How to Use Keyed-File Interface Calls in FORTRAN Programs 6-2
Processing Errors 6-3
Using FORTRAN Keyed-File Interface Calls in Other Languages 6-4

Keyed-File Interface Calls: Quick Reference 6-5
CLOSEM Call 6-6
DLTE Call 6-7
FILEDA Call 6-9
FILEIS Call 6-10
FLUSHM Call 6-12
GET Call 6-13
GETN Call 6-17
IFETCH Call 6-20
KEYLIST Call 6-22
KLCOUNT Call 6-26
KLSPACE Call 6-29
LOCKF Call 6-33
LOCKK Call 6-35
OPENM Call 6-38
PABORT Call 6-41
PBEGIN Call 6-42
PCOMMIT Call 6-42.2
PDETERM Call 6-42.3
PUT Call 6-43
PUTREP Call 6-45
REPLC Call 6-47
REWND Call 6-49
RMKDEF Call 6-50
RSBUILD Call 6-51
RSCLEAR Call 6-55
RSCLOSE Call 6-56
RSCOMB Call 6-57
RSDLTE Call 6-59
RSGETN Call 6-60
RSINFO Call 6-63
RSOPEN Call 6-65
RSPUT Call 6-67
RSREWND Call 6-68
RSSKIP Call 6-69
RSSTART Call 6-70
SKIP Call 6-71
STARTM Call 6-74
STOREF Call 6-76
UNLOCKF Call 6-78
UNLOCKK Call 6-79

Keyed-File Interface Calls 6

This chapter summarizes how to use keyed-file interface calls and includes a quick reference section of all keyed-file interface calls.

Use keyed-file interface calls to operate on keyed files. Five of the calls use FIT keywords and values to store information in a FIT and to retrieve information in a FIT. The calls are: IFETCH, FILEIS, FILEDA, OPENM, and STOREF. For more information on FIT keywords and values, see chapter 7, File Information Table Keywords and Values.

How to Use Keyed-File Interface Calls in FORTRAN Programs

This table summarizes the keyed-file interface calls and their purposes:

Call	Purpose
CLOSEM	Closes an open file
DLTE	Deletes a record
FILEDA	Creates a FIT for a direct-access file
FILEIS	Creates a FIT for an indexed-sequential file
FLUSHM	Copies the file in memory to disk
GET	Reads a record by its key value
GETN	Reads the next record in sequential order
IFETCH	Fetches a FIT value
KEYLIST	Fetches primary-key values from an alternate index
KLCOUNT	Fetches the number of primary-key values within a range in the alternate index
KLSPACE	Fetches the number of alternate-index blocks that contain the specified alternate-key value range
LOCKF	Locks a file
LOCKK	Locks a primary-key value
OPENM	Opens a keyed file
PABORT	Aborts a parcel
PBEGIN	Begins a parcel
PCOMMIT	Commits a parcel
PDETERM	Determines the state of a parcel
PUT	Writes a record
PUTREP	Writes or replaces a record
REPLC	Replaces a record
REWND	Positions the file at the lowest key value
RMKDEF	Creates an alternate key
RSBUILD	Builds a result set
RSCLEAR	Clears a result set
RSCLOSE	Closes an open result set
RSCOMB	Combines two result sets
RSDLTE	Deletes a key value from a result set
RSGETN	Uses the result set to get a record from the data file
RSINFO	Returns information about the result set
RSOPEN	Opens a result set
RSPUT	Adds a key value to a result set
RSREWND	Positions the result set at its beginning
RSSKIP	Positions the result set forward or backward
RSSTART	Positions the result set at a key value

Call	Purpose
SKIP	Repositions the file forward or backward a specified number
STARTM	Positions the file by a key value
STOREF	Stores a value in the FIT
UNLOCKF	Clears a file lock
UNLOCKK	Clears either a single primary-key value lock or all locks for the instance of open

If you are migrating a CYBER 170 FORTRAN program that uses GETNR or SEEKF calls, see appendix E, Differences Between NOS/VE FORTRAN and FORTRAN 5, for more information.

Each keyed-file interface call description lists the parameters for the call. The parameters must be specified in the indicated order.

Standard FORTRAN requires that all parameters be explicitly specified on a call. However, the keyed-file interface allows you to omit parameters whose values have been specified on previous calls.

To omit a parameter between two specified parameters, specify a zero (0) in the parameter position. To actually specify zero as a parameter value, you must store zero in a variable and specify the variable name on the call.

Unless indicated otherwise in the call description, a parameter value specified on a call is stored in the FIT so that it becomes the default value for subsequent calls.

Processing Errors

No type checking is performed on the values passed by a call. Passing an improper value could result in an internal routine detecting a computational fault such as an arithmetic overflow. To find the line that caused the error, use Debug to trace back the call chain.

To process errors, use IFETCH to retrieve the condition code from the FIT and then use CONDSYM to translate the condition code into the condition name.

In this example, the OPENM call attempts to open a new file. Assume the file exists so you receive an error whose condition code is 280263396310. CONDSYM translates the condition code to AA 3030.

```
CALL OPENM(fit_ptr, 'NEW', 'R')
CALL IFETCH(fit_ptr, '$ERROR_STATUS', condition_code)
IF (CONDITION_CODE .NE. 0) THEN
   CALL CONDSYM(condition_code, condition_name, length)
   PRINT *, 'Error encountered.  Condition name is ',condition_name
ENDIF
```

For more information on translating the $ERROR_STATUS value, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

For more information on a condition name, enter the online Diagnostic Messages manual and use the INDEX function on the condition name. This example shows how to enter the Diagnostic Messages manual:

```
/help manual=messages
```

Using FORTRAN Keyed-File Interface Calls in Other Languages

When a program written in a language other than FORTRAN or COBOL uses FORTRAN keyed-file interface calls, you must add the following object library to the program library list before executing the program:

```
AAF$4DD_LIBRARY
```

For example, the following SET_PROGRAM_ATTRIBUTES command adds the object library to the program library list.

```
/set_program_attributes  add_libraries=aaf$4dd_library
```

For more information about the program library list, see the manual NOS/VE Object Code Management Usage.

Keyed-File Interface Calls: Quick Reference

The following section describes all of the keyed-file interface calls in quick reference format.

You can check the status of any call (except for the PABORT, PBEGIN, PCOMMIT, and PDETERM calls) by retrieving the value of $ERROR_STATUS from the FIT for the specified call.

CLOSEM Call

Purpose Closes an open keyed file.

Format **CALL CLOSEM (fit, close_flag)**

Parameters **fit**

Variable containing the FIT pointer returned by the call that created the FIT.

close_flag

Close flag handling. Options are:

'U' or 'RET'

Detach the file.

'R'

Rewind the file. This option is provided for compatibility with earlier versions of the keyed file interface and is ignored. Use the OPENM call for file positioning.

'N'

Do not rewind or detach the file. This option is provided for compatibility with earlier versions of the keyed file interface and is ignored. Use the OPENM call for file positioning.

No default value is stored in the FIT for this parameter.

Remarks

- When a program finishes processing a keyed file, it should immediately call CLOSEM to close the file. Close processing copies any data or index blocks in memory to the mass storage file, updates internal tables, and writes statistics to the $ERRORS file (if requested by the $MESSAGE_CONTROL value). It also clears all locks for the instance of open.
- All files are closed at task termination. This is true whether the task terminates normally or abnormally.
- A CLOSEM call does not discard the FIT. The same FIT pointer variable can be specified on a subsequent OPENM call to open the same file again.
- CLOSEM should not be called for an instance of open that has a parcel in progress. Closing an instance of open to which a parcel applies aborts the parcel. For a description of parcels, see chapter 4, Parcels.

Examples This call closes and detaches a keyed file, preventing its further use in the program. The IFETCH call checks that the CLOSEM call completed successfully.

```
CALL CLOSEM (fit, 'U')
CALL IFETCH ('$ERROR_STATUS', status)
IF (status .NE. 0) CALL err_report
```

DLTE Call

Purpose Removes a record from a keyed file.

Format **CALL DLTE (fit, key_area, 0, 0, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

key_area

Location of the primary-key value of the record to be deleted.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

0

For FORTRAN 5 compatibility. New programs should set this parameter to zero.

0

Reserved position for unused parameter.

error_exit_procedure

Name of the error-exit procedure.

Remarks

- A DLTE call requires append, shorten, and modify access to the file. Otherwise, DLTE returns a nonfatal error.
- If the file could be shared (more than one instance of open could be changing the file at the same time), a record can be deleted only if the instance of open has a Preserve_Access_and_Content or Exclusive_Access lock on the primary-key value.

  A task can lock a primary-key value by calling LOCKK, GET, or GETN. To read about locks, see chapter 3, Sharing Keyed Files.
- You cannot delete a record by specifying its alternate-key value. You must specify its primary-key value. The key value specified on a DLTE call is processed as a primary-key value even if an alternate key is currently selected. A DLTE call deletes the primary-key value from all alternate indexes that reference it.
- DLTE searches for the primary-key value only in the nested file currently selected.
- If DLTE cannot find a record with the specified primary-key value, it returns a nonfatal error.
- A DLTE call does not change the file position or change the currently selected key or nested file.

For Better Performance

When deleting a sequence of records, it is most efficient to delete the records in order from the highest primary-key value to the lowest primary-key value. By working backwards, you can avoid relocation of records to be subsequently deleted.

- If a data block or index block contains no records as a result of the delete request, it is linked into a chain of empty blocks. These blocks are reused when new blocks are required for file expansion.

Examples

The DLTE call deletes the record with primary-key value ABCD. The IFETCH call checks that the DLTE call completed successfully.

```
key = 'ABCD'
CALL DLTE (fit, key)
CALL IFETCH (fit,'$ERROR_STATUS',status)
IF (status .NE. 0) CALL err_report
```

FILEDA Call

Purpose

Creates a file information table (FIT) for a direct-access file and, optionally, initializes FIT values.

Format

CALL FILEDA (fit, fit_keyword, fit_value, ..., *fit_keyword, fit_value*)

Parameters

fit

Integer variable in which the FIT pointer is returned.

fit_keyword

Character expression specifying a FIT keyword (must be followed by an allowable value for the attribute). The keyword must be a character expression (for example, '$KEY_LENGTH').

fit_value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description.

Remarks

- FIT keywords and values are described in chapter 7, File Information Table Keywords and Values.
- The FILEDA call must be the first call for a direct-access file because it creates the FIT for the file and sets the $FILE_ORGANIZATION value to DIRECT_ACCESS.

  All other calls for the file must specify the FIT pointer variable returned by the FILEDA call.
- Except for the $FILE_ORGANIZATION value, FILEDA call processing is the same as FILEIS call processing.

Examples

- This call creates a FIT for an existing direct-access file named MY_DA_FILE. It stores two FIT values: the local file name and the access modes.

```
 CALL FILEDA( fitptr, '$LOCAL_FILE_NAME', 'my_da_file',
+            '$ACCESS_MODE',    'READ,MODIFY')
```

- This call creates a FIT for a new direct-access file, specifying the minimum required attributes:

```
 CALL FILEDA( fitptr, '$LOCAL_FILE_NAME', 'my_da_file',
+            '$INITIAL_HOME_BLOCK_COUNT', 23,
+            '$KEY_LENGTH',               15,
+            '$MAXIMUM_RECORD_LENGTH',    80,
+            '$MINIMUM_RECORD_LENGTH',    15)
```

FILEIS Call

Purpose

Creates a file information table (FIT) for an indexed-sequential file and, optionally, initializes FIT values.

Format

CALL FILEIS (fit, fit_keyword, fit_value, ..., *fit_keyword, fit_value*)

Parameters

fit

Integer variable in which the FIT pointer is returned.

fit_keyword

Character expression specifying a FIT keyword (must be followed by an allowable value for the attribute). The keyword must be a character expression (for example, '$KEY_LENGTH').

fit_value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description.

Remarks

- For more information about FIT keywords and values, see chapter 7, FIT Keywords and Values.
- The FILEIS call must be the first call for an indexed-sequential file because it creates the FIT for the file and initializes the $FILE_ORGANIZATION value to INDEXED_SEQUENTIAL. All subsequent keyed-file interface calls must specify the variable containing the FIT pointer returned by the FILEIS call.
- The FILEIS call can specify any number of fit keyword and value pairs in any order. You can change FIT values specified by the FILEIS call using STOREF calls.
- FILEIS returns a nonfatal error ($ERROR_STATUS value AA2510) when it does not recognize a specified keyword. It also returns a nonfatal error ($ERROR_STATUS value AA2505) if a specified value is outside of the range applicable for the parameter.
- The FILEIS call associates the FIT with a local file name using the $LOCAL_FILE_NAME keyword. The old/new (ON) value indicates whether the file is a new or existing file.
- File attribute values specified by SET_FILE_ATTRIBUTE commands before program execution override corresponding attribute values specified by FILEIS calls.
- Attribute values in the FIT are checked for validity and consistency when the file is opened.

Examples

- This call creates a FIT for an existing indexed-sequential file named MY_IS_FILE. It stores two FIT values: the local file name and the access modes.

```
 CALL FILEIS(fitptr, '$LOCAL_FILE_NAME', 'my_is_file',
+            '$ACCESS_MODE','READ,MODIFY')
```

- This call creates a FIT for a new indexed-sequential file, specifying the minimum required attributes:

```
 CALL FILEIS(fitptr, '$LOCAL_FILE_NAME', 'my_new_is_file',
+            '$KEY_LENGTH',              15,
+            '$MAXIMUM_RECORD_LENGTH',   80,
+            '$MINIMUM_RECORD_LENGTH',   15)
```

FLUSHM Call

Purpose Writes all modified blocks to mass storage.

Format CALL FLUSHM (fit)

Parameters **fit**

Variable containing the FIT pointer returned by the call FILEIS or FILEDA that created the FIT.

Remarks

- A FLUSHM call requires append, shorten, or modify access to the file.
- A FLUSHM call ensures that the disk copy of the file contains the latest changes to the file. FLUSHM does not reposition or close the file. Blocks in memory are not disturbed.
- If the $FORCED_WRITE value in the FIT is TRUE, a data or index block is copied to disk immediately after the block is changed. However, a FLUSHM call also copies all internal file tables to disk, providing a complete backup copy.

Examples The STOREF call specifies the error-exit procedure to be called if the FLUSHM call detects an error. (The program has previously declared the ERREXIT subprogram as EXTERNAL.)

```
CALL STOREF (fit, '$ERROR_EXIT_PROCEDURE', errexit)
CALL FLUSHM (fit)
```

GET Call

Purpose Reads a record by its key value from an open keyed file.

Format **CALL GET (fit, working_storage_area, key_area, 0, major_key_length, 0, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

working_storage_area

Working storage area (location to which the record data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

---

key_area

Location of the key value of the record to be read.

0

For CYBER 170 compatibility. New programs should set this parameter to zero.

major_key_length

Major key length (in bytes); defaults to zero. It is reset to zero after the call.

When using a variable_length alternate key, a nonzero major key length value is required because it specifies the key-value length.

The parameter is ignored if the file is a direct-access file and its primary key is currently selected.

0

Reserved position for unused parameter.

error_exit_procedure

Error-exit procedure name.

Remarks

- A GET call requires at least read access to the file. To update file statistics, it also requires modify access.
- A GET call requires that a working storage area be specified on the call or in the FIT.
- GET searches for the specified key value in the currently selected nested file only.
- GET uses the primary or alternate key specified by the $KEY_NAME value in the FIT. The $KEY_NAME value is initially set to the primary key ($PRIMARY_KEY).

- When the primary key is selected, a key_area value must be specified on the call or in the FIT.

- When an alternate key is selected and the key_area values on the call and in the FIT are both zero, GET assumes that the alternate key value is in the working storage area at the position of the alternate key in the record.

  For example, if the alternate key is bytes 5 through 10 of the record, GET uses the contents of bytes 5 through 10 of the working storage area as the alternate-key value.

- The meaning of the major key length value depends on whether the selected key is fixed-length or variable-length.

  - For a fixed-length key, a nonzero major key length value specifies that GET is to search for the key using a major key. This means that, starting from the left of the key value, only major_key_length bytes of the key values are compared.

  - For a variable-length key, the major key length value specifies the key-value length. The key value is compared with the full key values stored in the index.

  A major key length value specified on a call is not stored in the FIT. A $MAJOR_KEY_LENGTH value specified in the FIT is cleared after it is used, so the program must specify a major key length value for each call that is to use a major key or a variable-length key value. (Major-key use is valid only while either the primary key of a indexed-sequential file or any alternate key is selected.) For more information, see the $MAJOR_KEY_LENGTH FIT keyword description.

- If an alternate key has been selected and the key is a concatenated key, the values for the pieces of the key must be assembled in the key area or the working storage area.

  In the key area, the pieces must be concatenated in the order defined for the alternate key.

  In the working storage area, the pieces must be stored in their fields in the record.

  For example, suppose the first piece of the alternate key is the third byte of the record and the second piece of the alternate key is the first byte of the record. To get the record whose first byte is an A and whose third byte is an *, either:

  - store A in the first byte of the working storage area and * in the third byte, or

  - store *A in the key area.

- GET searches for the first key value that satisfies the relation specified by the $KEY_RELATION value in the FIT.

  - If the relation is EQUAL_KEY and an equal key value does not exist in the file, GET returns a nonfatal error. The file is left positioned to read the next record (the record that would follow the specified record if it existed).

  - If the $KEY_RELATION value is GREATER_OR_EQUAL_KEY or GREATER_KEY and no key value in the file satisfies the relation, the data-exit (DX) procedure is called, if one is specified in the FIT. The file is left positioned at the end of information.

- If the $GET_AND_LOCK value in the FIT is -1 (YES), the GET call requests a lock on the primary-key value of the record to be read. The lock request uses the $AUTOMATIC_UNLOCK, $LOCK_INTENT, and $WAIT_FOR_LOCK values in the FIT. To read about locks, see chapter 3, Sharing Keyed Files.

  When an alternate key is selected, the GET call requests a lock on the first primary-key value in the key list only.

  If the GET call fails for any reason, it terminates without a lock on the primary-key value.

- The GET call reads data from the record until it reaches the end of the record or it has read the number of bytes specified as the working storage length in the FIT. (GET does not overwrite space following the working storage area with excess data.)

  If the record being read is longer than the working storage length, GET returns a nonfatal error.

- A successful GET call sets the record length value in the FIT to the actual length of the record. The record length value is not defined for an unsuccessful GET call.

- File positioning by a GET call differs depending on the file organization and the selected key:

  For a direct-access file with its primary key selected, the following statements are true:

  - GET does not change the file position used by GETN calls.

  - The only file position GET returns is end-of-record (16).

  - The only calls that can reposition the file are REWND and GETN. (STARTM is not valid.)

  - The $MAJOR_KEY_LENGTH and $KEY_RELATION values are not used.

  A GET call for a direct-access file with an alternate key selected is processed the same as a call to an indexed-sequential file with an alternate key selected.

- For an alternate key or an indexed-sequential file, the following statements are true:
  - At completion of a successful GET call, the file is positioned to read the record with the next highest key value. The file position returned can be end-of-record ($FILE_POSITION value 16) or, for an alternate key, end-of-key-list ($FILE_POSITION value 8).
  - An unsuccessful GET call returns a $FILE_POSITION value of 64 (end-of-information) in these cases:
    - The specified $KEY_RELATION was GREATER_THAN_OR_EQUAL and the key value was greater than all key values in the file.
    - The specified $KEY_RELATION was GREATER_THAN and the key value is the greatest in the file.
- A GET call that requests an unavailable lock leaves the file positioned to read the requested record.
- The program should call IFETCH to return the file position after a successful GET call.

  When the $FILE_POSITION value returned is 64 (end-of-information), the file is positioned at the end of the file and no GETN calls should be issued before file repositioning.
- GET can return the primary-key value of a record it found using an alternate-key value. If the $PRIMARY_KEY_ADDRESS value in the FIT is nonzero, GET returns the primary-key value in the $PRIMARY_KEY_ADDRESS location.

Examples This sequence of calls reads a record by major key value.

```
C Gets the first record whose key value begins with AB.

      KEY1 = 'ABCD'
      CALL GET (fit, record1, key1, 0, 2, 0, errexit)

C Gets the current file position and calls subroutine NOREC
C if no key value in the file begins with AB.  (The file
C would be left positioned at its end-of-information.)

      IF (IFETCH (fit, '$FILE_POSITION') .EQ. 64) THEN
         CALL norec
      ELSE

C Fetches the record length of record read and passes the
C record and its length to subroutine PROCDTA.

         CALL IFETCH (fit, '$RECORD_LENGTH', recleng)
         CALL procdta (record1, recleng)
      ENDIF
```

GETN Call

Purpose Reads the next record at the current file position.

Format **CALL GETN (fit, working_storage_area, key_area, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the call that created the FIT.

working_storage_area

Working storage area (location to which the record data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

key_area

Variable in which GETN returns the key value of the record.

For a variable-length alternate key, the key value is written to the variable followed by padding characters up to the maximum key length. The padding character used is the lowest character in the key-delimiter set.

For example, if the variable is 80 bytes long, the key value is 12 bytes, and the maximum key length is 31 bytes, the call first writes the 12-byte key value and then 19 padding characters. The GETN call does not write to the last 49 bytes.

error_exit_procedure

Error-exit procedure name.

Remarks

- A GETN call requires at least read access to the file. To update file statistics, it also requires modify access.
- A GETN call requires that a working storage area be specified on the call or in the FIT.
- If the $GET_AND_LOCK value in the FIT is -1 (YES), GETN requests a lock on the primary-key value of the record to be read. The lock request uses the $AUTOMATIC_UNLOCK, $LOCK_INTENT, and $WAIT_FOR_LOCK values in the FIT. To read about locks, see chapter 3, Sharing Keyed Files.

  If the GETN call fails for any reason, it terminates without a lock on the primary-key value.
- The GETN call reads data from the record until it reaches the end of the record or it has read the number of bytes specified as the working-storage length in the FIT. GETN cannot copy more data than the working-storage-area length.

- A successful GETN call sets the record-length value in the FIT to the actual length of the record. The record-length value is not defined for an unsuccessful GETN call.

- When an alternate key is selected, GETN calls return records in the key-value order provided by the alternate index.

  When the primary key of an indexed-sequential file is selected, GETN returns records in the key-value order provided by the primary index.

  However, no index exists for the primary key of a direct-access file so GETN does not return records in key-value order. It returns records in physical order by their location in the file.

  A GETN call that requests an unavailable lock leaves the file positioned to read the requested record.

- When a GETN call reads a record from the file, it returns a $FILE_POSITION value of 16 (or 8 if an alternate key is selected).

  After the GETN call that reads the last record in the file, the next GETN call returns a $FILE_POSITION of 64 (end-of-information). It returns an $ERROR_STATUS of 0 (no error), but no data or key values.

  A GETN call issued after a $FILE_POSITION value of 64 is returned, and before the file is repositioned, is an attempt to read beyond the end-of-information.

- The key value returned to the key_area location is the value of the currently selected key. If the selected key is an alternate key, the value returned is the alternate-key value.

  The length of the value returned is the key_length specified when the key was created. A variable-length alternate-key value is padded to its right with delimiter characters up to the maximum length for the key. (The padding character is the lowest character in the key-delimiter set.)

- GETN can also return the primary-key value when an alternate key is selected. If the $PRIMARY_KEY_ADDRESS value in the FIT is nonzero, GETN returns the primary-key value in the $PRIMARY_KEY_ADDRESS location.

Examples

This sequence of calls reads all records whose alternate key value is ABC into a very long character variable named $WORKING_STORAGE_ADDRESS.

```
      CALL STOREF (fit, '$KEY_NAME', 'ALT1')
      key = 'ABC'
      n = 1
      CALL GET (fit, working_storage_area(n), key, 0, 0, errexit)

      IF (IFETCH(fit, '$FILE_POSITION') .EQ. 8) THEN
        CONTINUE
      ELSEIF (IFETCH(fit, '$FILE_POSITION') .EQ. 16) THEN
10      n = n + IFETCH(fit,'$RECORD_LENGTH')
        CALL GETN (fit, working_storage_area(n), 0, 0)
        IF (IFETCH(fit, '$FILE_POSITION') .EQ. 16) GO TO 10
      ELSE
        CALL nodata
      ENDIF

      n = n + IFETCH(fit, '$RECORD_LENGTH')
      CALL procdta (working_storage_area, n)
```

IFETCH Call

Purpose

Retrieves a FIT value.

NOTE

IFETCH can be called as a function or as a subroutine.

Format

IFETCH (fit, fit_keyword)
or
CALL IFETCH (fit, fit_keyword, variable)

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

fit_keyword

Character expression specifying the FIT value to be fetched (such as, '$FILE_POSITION').

The keyword can be specified using uppercase and/or lowercase letters. Keywords are listed in chapter 7, File Information Table Keywords and Values.

variable

Variable to receive the FIT value.

Remarks

- Before a FIT is used to open a file, the only values that IFETCH can fetch from the FIT are those that have been stored in the FIT by the FILEIS or FILEDA call that created the FIT or by a STOREF call.
- While the file is open, IFETCH can fetch any value from the FIT. However, after the file is closed, IFETCH can only fetch certain values. The following is a list of the values that it can fetch.

Table 6-1. FIT Keywords That Can Be Fetched on a Closed File

$AUTOMATIC_UNLOCK	$KEY_RELATION
DX (data exit routine)	$LAST_OPERATION
$ERROR_EXIT_PROCEDURE	$LOCAL_FILE_NAME
$ERROR_STATUS	$LOCK_INTENT
$FILE_ORGANIZATION	$MAJOR_KEY_LENGTH
$FILE_POSITION	OC (opened/closed flag)
FNF (fatal/nonfatal flag)	ON (old/new flag)
$GET_AND_LOCK	$PRIMARY_KEY_ADDRESS
$GLOBAL_ACCESS_MODES	$WAIT_FOR_LOCK
$GLOBAL_SHARE_MODES	$WORKING_STORAGE_ADDRESS
$KEY_ADDRESS	$WORKING_STORAGE_LENGTH
$KEY_POSITION	

end_of_primary_key_list

Integer variable in which KEYLIST returns a value indicating whether the working storage area was long enough to contain all values in the requested range.

0 KEYLIST could not return all values in the requested range.

1 KEYLIST returned all values in the requested range.

transferred_byte_count

Integer variable which receives the total length, in bytes, of the primary-key values KEYLIST returned in the working storage area.

transferred_key_count

Integer variable which receives the number of primary-key values KEYLIST fetched.

file_pos

Integer variable in which the file position at completion of the KEYLIST call is returned.

Value	Meaning
8	The file is positioned at the end of a key list (positioned to fetch the first value in the next list).
16	The file is positioned at the end of a record, but not at the end of a key list (positioned to fetch the next value in the same key list).
64	The file is positioned at the end of the alternate index. (It cannot fetch any more values at this position.)

condition_code

Integer variable in which the condition code is returned. A zero value returned indicates successful completion.

For information on translating the condition code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- You must specify values for all KEYLIST parameters. KEYLIST does not use FIT values as default values.
- The program must select an alternate key before issuing a KEYLIST call.

- The high_key parameter value specifies the upper bound of the range of keys to be returned. The high_key_relation parameter indicates whether the primary-key values for the high_key value itself are returned.

  For example, suppose the high_key value is SMITH.

  - If you specify 'GREATER_KEY' as the high_key_relation value, KEYLIST returns the primary-key values for SMITH.
  - If you specify 'EQUAL_KEY' as the high_key_relation, KEYLIST does not return the primary-key values for SMITH. (It stops fetching values at the SMITH alternate-key value.)

- A major key consists of the leftmost bytes of a key. For a fixed-length key, a nonzero major_high_key parameter specifies the number of bytes of the high_key value KEYLIST is to use as a major key. A major key search compares only the leftmost bytes of the key values on the call and in the index.

  For example, suppose the high_key value is ABCDEF and the major_high_key parameter value is 2. The major key used is AB. KEYLIST returns primary-key values until it finds an alternate-key value beginning with the characters AB or higher. Whether it returns the primary-key values for the AB value depends on the high_key_relation parameter value.

  Major_key use is invalid when the primary key of a direct-access file is selected.

- The KEYLIST call could return the same primary-key value more than once if the primary-key value is associated with more than one alternate-key value. This is possible if the repeating-groups attribute is defined for the alternate key.

- KEYLIST returns primary-key values until it reaches the end of the specified range or until it cannot fit another value into the working storage area. By checking the end_of_primary_key_list value, the program can determine if all requested values were returned and, if not, call KEYLIST again to fetch the rest of the values.

- KEYLIST repositions the file as it fetches key values. At completion of the call, the file is positioned at the end of the last key value returned and positioned to continue fetching values at that point if KEYLIST is called again.

- KEYLIST cannot be called for an instance of open that has a parcel in progress. For a description of parcels, see chapter 4, Parcels.

Examples

- These calls fetch all primary-key values in the alternate index. The STOREF call selects alternate key ALT_KEY_1 and positions the file at the beginning of the alternate index. The subroutine KEYPROC processes the key values fetched. The KEYLIST call is repeated until all primary-key values are fetched.

```
      CALL STOREF (fit, '$KEY_NAME', 'ALT_KEY_1')

   10 CALL KEYLIST(fit, 0, 0, 'HIGHEST_KEY', working_storage_area,
     +     LEN(working_storage_area), keyend, length, keycnt,
     +     filpos, ccode)

      IF (ccode .NE. 0) THEN
         CALL errprog
      ELSE
         CALL keyproc(working_storage_area,
     +        LEN(working_storage_area), length, keycnt)
      ENDIF

      IF (keyend .EQ. 0) GO TO 10
```

- The STARTM call positions the alternate index at alternate-key value ABCD. The KEYLIST call then fetches the primary-key values for that alternate-key value.

```
    keyval='ABCD'
    CALL STARTM(fit, keyvalue)
    CALL KEYLIST(fit, keyvalue, 0, 'GT', big_array, LEN(big_array),
   +            keyend, length, keycount, file_pos, ccode)
    IF (ccode .NE. 0) CALL errprog
```

KLCOUNT Call

Purpose Counts the number of primary-key values associated with the specified range of alternate-key values in the alternate index.

Format **CALL KLCOUNT (fit, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, list_count_limit, list_count, condition_code)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

low_key

Alternate-key value at which the range begins. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major_key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

low_key_relation

Indicates where KLCOUNT is to start counting primary-key values. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Start at the lowest alternate-key value greater than the low_key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Start at the lowest alternate-key greater than or equal to the low-key value.

'LOWEST_KEY' or 'LK'

Start counting at the beginning of the alternate index. (The low_key and major_low_key values are ignored when 'LOWEST_KEY' is specified.)

high_key

Alternate-key value at which the range ends. The value must be valid for the key type (integer for an integer key, character for a collated or uncollated key).

major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

high_key_relation

Indicates when KLCOUNT is to stop counting primary-key values. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Stop at the lowest alternate-key value greater than the high-key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Stop at the lowest alternate-key value greater than or equal to the high-key value.

'HIGHEST_KEY' or 'HK'

Stop at the end of the alternate index. (The high_key and major_high_key values are ignored when 'HIGHEST_KEY' is specified.)

list_count_limit

Maximum number of primary-key values counted. If you specify zero for the parameter, no limit is set.

list_count

Integer variable in which the primary-key value count is returned.

condition_code

Integer variable in which the condition code is returned. A zero value returned indicates successful completion.

For information on deciphering the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

To determine the meaning of a nonzero condition code, see the Diagnostics Messages for NOS/VE manual.

Remarks

- You must specify values for all KLCOUNT parameters. KLCOUNT does not use default FIT values.
- The program must select an alternate key before issuing a KLCOUNT call.
- The low_key and high_key parameter values specify the lower and upper bounds, respectively, of the range to be counted.
- The low_key_relation and high_key_relation parameters indicate whether the primary-key values for the low_key values are included in the count, and whether the primary_key values for the high_key values are excluded from the count.

  For example, suppose the low_key value is JONES and the high_key value is SMITH.

  - If you specify 'GREATER_KEY' as the low_key_relation value, KLCOUNT does not count the primary-key values for JONES.

- If you specify 'EQUAL_KEY' as the low_key_relation value, KLCOUNT counts the primary-key values for JONES.
- If you specify 'GREATER_KEY' as the high_key_relation value, KLCOUNT counts the primary-key values for SMITH.
- If you specify 'EQUAL_KEY' as the high_key_relation value, KLCOUNT does not count the primary-key values for SMITH.

- A major key consists of the leftmost bytes of a key. For a fixed-length key, a nonzero major_high_key or the major_low_key parameter specifies the number of bytes of the high_key or low_key value, respectively, that KLCOUNT is to use as a major key. A major key search compares only the leftmost bytes of the key values on the call and in the index.

  For example, suppose the low_key value is ABCDEF. If the major_low_key parameter value is 2, the major key used is AB. KLCOUNT would then search for the lowest alternate-key value whose first two characters are greater than or equal to AB.

- The KLCOUNT call could count the same primary-key value more than once if the primary-key value is associated with more than one alternate-key value. This is possible if the repeating-groups attribute is defined for the alternate key.

- The list_count_limit parameter can minimize the processing required for the call.

  For example, if you call KLCOUNT to determine whether the number of primary-key values is 0, 1, or more than 1, you should set the list_count_limit to 2.

- KLCOUNT cannot be called for an instance of open for which a parcel is in progress, For a description of parcels, see chapter 4, Parcels.

Examples

- These calls return the number of primary-key values for alternate key ALT_KEY_1 in the integer variable KEYCOUNT. The completion code is returned in the integer variable CONDITION_CODE.

```
      CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
      CALL KLCOUNT(fit, 0, 0, 'LOWEST_KEY',
     +                 0, 0, 'HIGHEST_KEY', 0, keycount,
     +                 condition_code)
      IF (condition_code .NE. 0) CALL errprog
```

- These calls return the number of primary-key values associated with alternate-key values that begin with 'C' (the major-key value).

```
      CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
      CALL KLCOUNT(fit, 'C', 1, 'EQ', 'C', 1, 'GT', 0,
     +            keycount, condition_code)
      IF (condition_code .NE. 0) CALL errprog
```

KLSPACE Call

Purpose

Returns the number of alternate-index blocks that contain the specified range of alternate-key values.

Format

CALL KLSPACE (fit, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, block_count, block_space, condition_code)

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

low_key

Alternate-key value at which the range begins. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

low_key_relation

Indicates whether the low_key value is included in the range. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Exclude the low_key value from the range.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Include the low_key value in the range.

'LOWEST_KEY' or 'LK'

The range starts at the beginning of the alternate index. (The low_key and major_low_key values are ignored when 'LOWEST_KEY' is specified.)

high_key

Alternate-key value at which the range ends. The value must be valid for the key type (integer for an integer key, character for a collated or uncollated key).

major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

high_key_relation

Indicates where the range ends in relation to the highest value in the range. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Include the high_key value in the range.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Exclude the high_key value from the range.

'HIGHEST_KEY' or 'HK' or 'HIGHEST_KEY'

The range ends at the end of the alternate index. (The high_key and major_high_key values are ignored when 'HIGHEST_KEY' is specified.)

block_count

Integer variable in which the block count is returned.

block_space

Integer variable in which the combined length of the blocks (in bytes) is returned (the block count multiplied by the block size).

condition_code

Integer variable in which the condition code is returned. A zero value returned indicates successful completion.

For information on deciphering the $ERROR_STATUS value, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

You can look up the meaning of any nonzero condition code in the Diagnostic Messages manual.

Remarks

- You must specify values for all KLSPACE parameters. KLSPACE does not use FIT values as default values.
- An alternate key must be the currently selected key when KLSPACE is called.
- The low_key, major_low_key, low_key_relation, high_key, major_high_key, and high_key_relation parameters specify the range of alternate-key values. Their use on a KLSPACE call is the same as on a KLCOUNT call. For details, see the Remarks in the KLCOUNT call description.
- A KLSPACE call does not actually find the specified alternate-key values in the alternate index. Rather, it searches the index to determine the number of blocks at the lowest level that would contain the specified range of alternate-key values.

  (An alternate index is an indexed-sequential structure with one or more index levels. The lowest level of blocks actually contain the alternate-key values and their corresponding primary-key values.)

- KLSPACE returns a value even if the specified low_key and high_key values are not in the alternate index. It returns the number of blocks that would contain the range if the values existed in the index.

- An accurate primary-key value count (such as that returned by KLCOUNT) cannot be derived from the block count that KLSPACE returns. The block counts for ranges containing the same number of primary-key values could differ because the ranges can span blocks.

  For example, suppose a range contains only one alternate-key value. If the record for the alternate-key value spans two blocks, the block count returned is 2, not 1.

- Because a KLSPACE call is faster than a KLCOUNT call, it can be used for a quick comparison of the relative lengths of primary-key lists (see the KLSPACE Example).

- The block_length value that KLSPACE returns can be used when comparing primary-key lists for files with different block sizes. Larger blocks require longer searches.

Examples

Assume that a program is to find a set of records in response to this query:

Find the Jones on Madison Avenue with more than two dependents.

Assume that Jones is a value for alternate key ALT_KEY_1 and Madison Avenue is a value for alternate key ALT_KEY_2. The number of dependents is not an alternate key so the program must read the data records to find that information.

The program could read the set of records for either Jones or Madison Avenue. To minimize the number of records read, the program first issues KLSPACE calls to compare the two primary-key value lists.

The following call sequence gets the block count values, compares them, and then stores the alternate-key name and value to be used.

```
 CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')

 CALL KLSPACE(fit, 'Jones', 0, 'EQ', 'Jones',
+ 0, 'GT', block_count1, block_length, condition_code)
 IF (condition_code .NE. 0) CALL errprog

 CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_2')

 CALL KLSPACE(fit, 'Madison Avenue', 0, 'EQ',
+ 'Madison Avenue', 0, 'GT', block_count2, block_length,
+  condition_code)
 IF (condition_code .NE. 0) CALL errprog

 IF (block_count1 .GE. block_count2) THEN
   keyval='Madison Avenue'
 ELSE
   keyval='Jones'
   CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
 END IF
```

LOCKF Call

Purpose Requests a file lock.

Format **CALL LOCKF (fit, wait_for_lock, lock_intent)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT. It specifies the instance of open to be locked.

wait_for_lock

Specifies whether the task waits if the lock is not immediately available. Options are:

'YES'

Task waits until either the lock is available or a time period has passed. The default is 60 seconds. When the time period has passed, LOCKF returns a nonfatal condition code.

'NO'

LOCKF terminates, and the lock is unavailable.

If 0 is specified as the wait_for_lock value on the call, the FIT value $WAIT_FOR_LOCK is used. The default $WAIT_FOR_LOCK value is YES.

The task does not wait if a deadlock exists.

lock_intent

Specifies the lock intent. You can specify the lock intent using uppercase and/or lowercase letters. For more information, see Lock Intents in chapter 3, Sharing Keyed Files. Options are:

'Exclusive_Access' or 'EA'

Only the instance of open can access records in the nested file. All other requests by other instances of open are denied.

'Preserve_Access_and_Content' or 'PAC'

Only the instance of open can update records in the nested file. All instances of open can read the file. All instances of open can have Preserve_Content locks, but all instances of open are denied Exclusive_Access or Preserve_Access_and_Content locks.

'Preserve_Content' or 'PC'

If the file is not shared, the lock owner can update the records in the nested file. No other updates are allowed. All instances of open can have Preserve_Content locks, and one Preserve_Access_and_Content lock can exist for each primary-key value and for the file as a whole. All Exclusive_Access lock requests are denied.

If you specify the lock intent as 0, the default FIT value for $LOCK_INTENT, Preserve_Access_and_Content, is used.

Remarks

- The lock applies to the current nested file only, as specified by the $NESTED_FILE_NAME value.
- You can change the maximum waiting period for the lock. The default value is 60 seconds.

  To change the waiting period, create a NOS/VE integer variable named AAV$RESOLVE_TIME_LIMIT and assign it the waiting period value in seconds. The timeout period should not exceed the LOCK_EXPIRATION_TIME attribute value.

  For example, this call executes a NOS/VE command that sets the waiting period at 45 seconds.

  ```
   CALL SCLCMD ('create_variable, name=AAV$RESOLVE_TIME_LIMIT,
  +  kind=integer, value=45')
  ```

  Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks in the job, specify SCOPE=JOB on the CREATE_VARIABLE command.
- Assuming the LOCK_EXPIRATION_TIME file attribute is nonzero, the lock could expire. LOCKF returns a nonfatal condition code value if the expired lock prevents granting of the requested lock. To read about lock expiration, see Lock Expiration and Clearing in chapter 3, Sharing Keyed Files.
- File locks cannot be automatically unlocked. To clear a single file lock, call UNLOCKF. To clear all file locks for an instance of open, call CLOSEM or UNLOCKK with the 'ALL' option.
- LOCKF cannot be used to request a lock intent of 'PAC' or 'PC' for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

Examples

This call requests a file lock. The wait_for_lock and lock_intent parameter values are supplied by the default $WAIT_FOR_LOCK and $LOCK_INTENT FIT values. The default value for $WAIT_FOR_LOCK is YES and the default value for $LOCK_INTENT is Preserve_Access_and_Content.

```
CALL LOCKF (fit)
```

LOCKK Call

Purpose Requests a lock on a primary-key value.

Format **CALL LOCKK (fit, key_area, wait_for_lock, automatic_unlock, lock_intent)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

key_area

Location containing the primary-key value to be locked.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

---

wait_for_lock

Indicates whether the task waits if another task has a conflicting lock on the primary-key value and no deadlock exists. Options are:

'YES'

Task waits until either the lock is available or the wait time period has passed. The default is 60 seconds. When the time period has passed, LOCKK returns a nonfatal condition code.

'NO'

LOCKK terminates, returning a nonfatal condition code, indicating that the lock is unavailable.

If you specify the wait_for_lock value as 0, the default FIT value for $WAIT_FOR_LOCK is YES.

automatic_unlock

Indicates whether automatic unlock is used for this lock. Options are:

'YES'

Automatic unlock is used. The lock is cleared when one of the following occurs:

- The task issues a request for another record.
- The task issues a non-update request for the same record.
- The task completes an update request specifying the locked key value.

'NO'

Automatic unlock is not used.

If you specify the automatic_unlock value as 0, the default FIT value for $AUTOMATIC_UNLOCK (YES) is used.

NOTE

Automatic unlock cannot be used with Preserve_Content lock intent.

lock_intent

Lock intent. You can specify lock intent using uppercase and/or lowercase letters. Options are:

'Exclusive_Access' or 'EA'

Only the instance of open can access records in the nested file. All other requests by other instances of open are denied.

'Preserve_Access_and_Content' or 'PAC'

Only the instance of open can update records in the nested file. All instances of open can read the file. All instances of open can have Preserve_Content locks, but all instances of open are denied Exclusive_Access or Preserve_Access_and_Content locks.

'Preserve_Content' or 'PC'

If the file is not shared, the lock owner can update the records in the nested file. No other updates are allowed. All instances of open can have Preserve_Content locks, and one Preserve_Access_and_Content lock can exist for each primary-key value and for the file as a whole. All Exclusive_Access lock requests are denied.

For more information, see Lock Intents in chapter 3, Sharing Keyed Files. If you specify the lock intent value as 0, the default FIT value for $LOCK_INTENT (Preserve_Access_and_Content) is used.

Remarks

- LOCKK only locks primary-key values. Even if an alternate key is currently selected, the key value in the specified key area is assumed to be a primary-key value.
- A LOCKK call can reserve a presently unused primary-key value for subsequent use by the task.
- A LOCKK call does not verify that the key value is valid, nor does it check whether the key value is already in the file. The key value is verified by a subsequent call that uses the key value.
- Assuming the LOCK_EXPIRATION_TIME file attribute is nonzero, the lock could expire. LOCKK returns a nonfatal condition code value if the expired lock prevents granting of the requested lock. To read about lock expiration, see Lock Expiration and Clearing in chapter 3, Sharing Keyed Files.

- You can change the maximum waiting period for the lock. The default is 60 seconds.

  To change the waiting period, create a NOS/Ve integer variable named AAV$RESOLVE_TIME_LIMIT and assign it the waiting period value in seconds.

  For example, this call executes a NOS/VE statement that sets the waiting period at 45 seconds.

```
CALL SCLCMD ('var AAV$RESOLVE_TIME_LIMIT: integer=45; varend')
```

  Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks in the job, specify JOB as the scope on the VAR/VAREND statement.

- LOCKK returns a nonfatal error if the requested lock could cause a deadlock. A potential deadlock can be detected only if the wait_for_lock value for the call is YES.

  To clear the deadlock situation, the task should clear its locks. It can then request the locks again.

- Besides the automatic unlock, a task can unlock a key value by calling UNLOCKK or by closing the instance of open.

Examples

- This call requests a lock on a key value. The key_area, wait_for_lock, automatic_unlock, and lock_intent values are supplied by these default FIT values:

FIT Keyword	Default FIT Value
$KEY_ADDRESS	No default.
$WAIT_FOR_LOCK	YES
$AUTOMATIC_UNLOCK	YES
$LOCK_INTENT	Preserve_Access_and_Content

```
CALL LOCKK(fit)
```

- This call requests a lock on the key value in variable KEY1. The next call writes the record. The lock is automatically unlocked at completion of the write request.

```
CALL LOCKK(fit, key1, 'YES', 'YES', 'Exclusive_Access')
CALL PUTREP(fit, array1, 15, key1)
```

OPENM Call

Purpose Opens a keyed file.

Format **CALL OPENM (fit, open_option, file_position)**

Parameters **fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

open_option

Type of processing. This parameter is optional. If you omit the open_option parameter, the default is 0. Options are:

0

Open the file using the access and share modes specified in the FIT. If no access and share modes have been stored in the FIT, the file is opened with read access and no sharing. If you omit the open_option parameter, the default is 0.

'INPUT'

Open the file for reading only. File statistics are not kept.

'OUTPUT'

Open the file for writing only.

'I-O' or 'IO'

Open the file for reading and writing.

'NEW'

A new file is being created. The $ACCESS_MODE FIT value is set to 'OUTPUT' and the old/new (ON) FIT value is set to 'NEW'.

If you specify 'INPUT', 'OUTPUT', 'IO', 'I-O', or 'NEW' on the OPENM call, the access and share mode values in the FIT are ignored. The access and share modes set by these options are:

Open_option	Access Modes Set	Share Modes Set
'INPUT'	Read	Read
'OUTPUT'	Modify, shorten, append	None
'IO'	Read, modify, shorten, append	None
'NEW'	Read, modify, shorten, append	None

file_position

File positioning when the file is opened. This parameter is optional. If you omit the file_position parameter, the default is 0. Options are:

0

Use the $OPEN_POSITION value in the FIT.

'R'

Rewind the file (position the file to read the record with the lowest key value). This is the default if the $OPEN_POSITION value in the FIT is zero.

'E'

Position the file after the record with the highest key value. (A GETN call at this position would return end-of-information (EOI) status.)

Remarks

- The OPENM call to open a keyed file must precede all other keyed-file interface calls except FILEDA, FILEIS, IFETCH, and STOREF calls.
- When opening an existing file, the old/new (ON) value in the FIT or on the call must be 'OLD'. Similarly, when opening a new file, the old/new (ON) value in the FIT or on the call must be 'NEW'.
- The access modes requested when the file is opened determine the processing allowed on the file. For example, if you specify 'INPUT' on the OPENM call, you cannot call PUT to write a record to the file.
- An existing file must be attached with the appropriate usage mode set for the type of processing (read permission for 'INPUT', write permissions for 'OUTPUT', or read and write permissions for 'I-O').
- Multiple instances of open are allowed for a file. Each instance of open must have its own FIT. So before the program attempts to open an already open file, it must call FILEIS or FILEDA to create another FIT.
- An OPENM call performs these steps:
  1. OPENM checks the old/new (ON) flag in the FIT to determine if the file is a new file or an existing file.
     a. If the file is a new file, OPENM creates the file using the file name specified by the $LOCAL_FILE_NAME value in the FIT.
     b. If the file is an existing file, OPENM searches for the file in the current working catalog using the file name specified by the $LOCAL_FILE_NAME in the FIT.

2. OPENM initializes file attribute values in the FIT as follows:

   a. If the file is an existing file, OPENM verifies attribute values stored in the FIT against the corresponding attribute values preserved with the file. If the program has not stored a FIT value for a preserved attribute, OPENM copies the attribute value preserved with the file to the FIT.

   b. If SET_FILE_ATTRIBUTES commands specified one or more attribute values for the file before the program began, OPENM overwrites the corresponding values in the FIT. Only temporary attribute values can be specified for an existing file.

3. OPENM checks that the FIT contains appropriate values for the keyed-file organization. It also checks that the values are consistent.

4. OPENM positions the file according to the $OPEN_POSITION value.

5. OPENM loads the collation-table module if the $KEY_TYPE value is COLLATED. The entry point name used is the $COLLATE_TABLE_NAME FIT value.

6. It also loads the error-exit procedure if a value has been stored for $ERROR_EXIT_PROCEDURE_NAME.

7. OPENM sets the open/closed (OC) flag in the FIT to open.

PABORT Call

Purpose Aborts a file-spanning parcel or a file-level parcel.

Format **CALL PABORT (system_parcel_name, condition_code, message_area)**

Parameters

system_parcel_name

Character variable containing the name returned by the PBEGIN call that began the parcel. The system_parcel_name is 31 characters.

If the system_parcel_name is the name of a file-level parcel, no messages are stored in the parcel log. In this case, you cannot access the parcel log or fetch the parcel state, and the message_area parameter is ignored.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

message_area

Character variable containing data to be stored in the parcel log record written by this call. It can be fetched later by a PDETERM call.

The length of the character variable determines the length of the message.

If this parameter is omitted, no message is stored in the parcel log. If the system_parcel_name specifies a file-level parcel, this parameter is ignored.

Remarks

- Only the task that began the parcel or the system can abort the parcel with PABORT.
- A parcel commit is transformed into a parcel abort if the commit is not successful.
- When an instance of open to which the parcel applies is closed, any file-level parcel in progress for that instance of open is immediately aborted. The abort of the file-level parcel causes the entire file-spanning parcel to abort when the task attempts to commit the parcel.
- The call stores no information in any FIT. The program must check the condition_code returned to determine if the call completed successfully.
- For more information, see the description of Parcels in chapter 4, Parcels.

PBEGIN Call

Purpose Marks the beginning of a file-spanning parcel or a file-level parcel.

Format **CALL PBEGIN (user_parcel_name, fit_list, number_of_fits, condition_code, system_parcel_name, log, message_area)**

Parameters

user_parcel_name

Character variable containing the user-defined name for the new parcel. The user_parcel_name is 1 to 31 characters.

If the user_parcel_name is to be used later to reference the parcel, the name should be unique for all parcels recorded on the parcel log. The program could call the CYBIL system procedure PMP$GENERATE_UNIQUE_NAME to create the parcel name (see the CYBIL System Interface manual).

fit_list

Integer array of FIT pointers specifying the instances of open to which the parcel applies.

To begin a file-level parcel, you must specify only one pointer.

(The FIT pointer is returned by the call that created the FIT. The pointer must have been specified previously on an OPENM call that opened a keyed file.)

number_of_fits

Number of FIT pointers specified by the fit_list parameter.

If a 1 is specified, and the log and message_data parameters are not specified, PBEGIN begins a file-level parcel instead of a file-spanning parcel.

If a 0 is specified, PBEGIN begins a file-spanning parcel and specifies that the parcel applies to *all* keyed files that have parcels enabled and are open when the parcel is begun. When 0 is specified, the fit_list parameter is ignored.

condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on translating the integer returned, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

system_parcel_name

Character variable in which the name the system assigns to the parcel is returned. The system_parcel_name is 31 characters. This name should be specified on later calls for the parcel.

log

Character variable containing the catalog path name of the log to be used for this parcel. This parameter is used only for file-spanning parcels, not for file-level parcels.

If the specified character variable contains only blanks, the default log is used. The default parcel log is the log specified by the SCL variable AAV$LOG_SELECTION, which is initially set by the system prolog, or the default system log, $SYSTEM.AAM.SHARED_RECOVERY_LOG.

If this parameter is omitted, the default parcel log is used.

message_area

Character variable containing data to be stored in the parcel log record written by this call. It can be fetched later by a PDETERM call. This parameter is used only for file-spanning parcels, not for file-level parcels.

The length of the character variable determines the maximum length of the message returned. Longer messages are truncated.

If this parameter is omitted, no message is stored in the parcel log.

Remarks

- To begin a file-level parcel, specify only the first five parameters of this call, and specify the value of 1 for the number_of_fits parameter.

  Otherwise, a file-spanning parcel is begun.

- A FORTRAN task can have only one file-spanning parcel in progress at a time. Therefore, a second PBEGIN call cannot be issued until the current parcel has been committed or aborted, unless the second call is for a file-level parcel for an instance of open not included in a file-spanning parcel.

- The call stores no information in the FIT. The program must check the condition code returned to determine if the call completed successfully.

- For more information, see chapter 4, Parcels.

PCOMMIT Call

Purpose

Commits a file-spanning parcel or a file-level parcel, making its update operations permanent.

Format

CALL PCOMMIT (system_parcel_name, condition_code, message_area)

Parameters

system_parcel_name

Character variable containing the name returned by the PBEGIN call that began the parcel. The system parcel name is 31 characters.

If the system_parcel_name is the name of a file-level parcel, no messages are stored in the parcel log. In this case, you cannot access the parcel log or fetch the parcel state, and the message_area parameter is ignored.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the integer returned, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

message_area

Character variable containing data to be stored in the parcel log record written by this call. It can be fetched later by a PDETERM call.

The length of the character variable determines the length of the message.

If this parameter is omitted, no message is stored in the parcel log. If the system_parcel_name parameter specifies a file-level parcel, this parameter is ignored.

Remarks

- If the parcel commit is not successful, the system calls PABORT to abort the parcel. An appropriate status is returned as the condition_code and the specified message (if any) is stored in the abort record in the parcel log.
- The call stores no information in the FIT. The program must check the condition code returned to determine if the call completed successfully.
- For more information, see chapter 4, Parcels.

PDETERM Call

Purpose Returns information on the current state of a file-spanning parcel.

Format **CALL PDETERM (parcel_name, name_type, state, condition_code, log, direction, days, hours, minutes, message_area, returned_length)**

Parameters

parcel_name

Character variable containing the parcel name. It can be the system-defined name or the user-defined name.

NOTE

Use the system-defined name whenever it is available. This is because the user-defined name may not be a unique name.

name_type

Integer indicating whether the parcel specified on the call is the system-defined name for a file-spanning parcel, the user-defined name for a file-spanning parcel, or the name of a file-level parcel, as follows:

0 System-defined name for a file-spanning parcel.

1 User-defined name for a file-spanning parcel.

2 Name of a file-level parcel.

state

Integer variable in which the call returns one of the following values:

0 Parcel active. The call found a begin record for the parcel on the log, but no commit or abort record. The call returns the message it finds in the begin record, if any.

1 Parcel committed. The call found a commit record for the parcel on the log. The call returns the message it finds in the commit record, if any. If no message is found, the call returns the message, if any, from the begin record.

2 Parcel aborted by system. The call found an abort record for the parcel on the log. The call returns the message it finds in the abort record, if any. If no message is found, the call returns the message, if any, from the begin record,

3 Parcel aborted by user. The call found an abort record for the parcel on the log. The call returns the message it finds in the abort record, if any.

4 Parcel not found. The call found no records for the specified parcel in the log. Check to make sure the correct log is specified.

5 Parcel indeterminate. The call may have found a begin record for the parcel, but also found indication of a catastrophic, unrecoverable error that prevented completion of the parcel. Check to make sure that a log exists or that the task has access to the log.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the integer returned, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

log

Character variable containing the catalog path name of the log to be searched. The name must be the same name specified on the log parameter on the PBEGIN call.

If the specified character variable contains only blanks, the default log is used. The default parcel log is the log specified by the SCL variable AAV$LOG_SELECTION, which is initially set by the system prolog, or the default system log, $SYSTEM.AAM.SHARED_RECOVERY_LOG.

direction

Integer specifying the direction in which the log is searched. Options are:

'BACKWARD' or 'B'	The log is searched backward from the time specified by this call.
'FORWARD' or 'F'	The log is searched forward from the time specified by this call.

If this parameter is omitted, the log is searched backward.

days

Number of days subtracted from the current time when determining the initial time for the log search (integer from 0 through 31).

The default is 0.

hours

Number of hours subtracted from the current time when determining the initial time for the log search (integer from 0 through 23).

The default is 0.

minutes

Number of minutes subtracted from the current time when determining the initial time for the log search (integer from 0 through 59).

The default is 0.

message_area

Character variable in which the message stored in the log record is returned. The message returned is the last message that was written to the parcel log. The message could have been written by a PABORT, PBEGIN, or PCOMMIT call.

The length of the character variable determines the maximum length of the message returned. Longer messages are truncated.

This parameter can be omitted only if the returned_length parameter is omitted. If omitted, no message is returned.

returned_length

Integer variable in which the call returns the length (in bytes) of the message actually returned in the message_area variable. If the message in the record is longer than the message_area variable, an error is returned.

If this parameter is omitted, no message length is returned.

Remarks

- Any task can get information about a parcel if it:
  - Knows the parcel name, and
  - Knows the catalog path name to be searched, and
  - Is running in a ring at least as privileged as the ring from which the PBEGIN call was issued.
- The call searches the specified parcel log to find the commit, abort, or begin record for the parcel. It returns the message stored in the log record, if any.
- The specified days, hours, and minutes are subtracted from the current time to set the starting point of the log search.

  For example, if 2 days, 2 hours, and 30 minutes are specified and the current time is 1:02 p.m. on August 28, the starting point is 10:32 a.m. on August 26. The search proceeds in the direction specified by the direction parameter.
- If you believe the parcel record is after the specified time, specify 'FORWARD' for the search_direction so the search is from the specified time forward to the current time.

  Otherwise, if you believe the parcel record is before the specified time, specify 'BACKWARD' for the direction so the search is from the specified time backward.
- If the name-type parameter contains the value of 2, only the states 0 or 4 are returned.
- For more information, see chapter 4, Parcels.
- The call stores no information in the FIT. The program must check the condition code returned to determine if the call completed successfully.

This page intentionally left blank.

PUT Call

Purpose Writes a record to a keyed file.

Format **CALL PUT (fit, working_storage_area, record_length, key_area, 0, 0, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

working_storage_area

Location from which data is copied to the file.

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

record_length

Record length in bytes. The parameter is used if the record type is variable length; it is ignored if the record type is fixed-length.

key_area

Location containing the primary key value of the new record. This parameter is ignored for files with embedded keys.

0

For CYBER 170 compatibility. New programs should set this parameter to zero.

0

Reserved position for an unused parameter.

error_exit_procedure

Error-exit procedure name.

Remarks

- A PUT call requires at least append access as indicated by the $ACCESS_MODE or $ACCESS_AND_SHARE_MODES value in the FIT. If alternate keys are defined in the file, a PUT call requires append, shorten, and modify access in order to update the alternate indexes.
- Before the program calls PUT, it must store the record data in the working-storage area. If the primary key is nonembedded, it must also store the key value in the key area.
- The specified primary-key value must not already exist in the file.
- You always specify a primary-key value on a PUT call, not an alternate-key value, even if an alternate key is currently selected.

- The PUT call updates each alternate index that is to include the new record. If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the PUT call returns a nonfatal error.
- If the file has fixed-length records, the record_length value on the call (and in the FIT) is ignored. The length of the record written is always the fixed record length for the file.

  A warning message is issued for the first PUT, PUTREP, or REPLCE call whose record_length value differs from the fixed record length for the file. If the record_length is less than the fixed record length, data is not padded so unknown data could be written as the last part of the record. If the record_length is longer, excess data is truncated.

For Better Performance

When writing to an indexed-sequential file, the program should write records in order by ascending key values. This results in faster execution and a more efficient file structure. Your program could write the records to a sequential file and then call Sort/Merge to sort and write the records to an indexed-sequential file.

- A PUT call returns an error when it cannot write the record because the file has reached a limit. Limits on the file include:
  - The number of records in the file cannot exceed the $RECORD_LIMIT value.
  - In an indexed-sequential file, the number of index levels cannot exceed 15.
  - The number of bytes of file space cannot exceed the FILE_LIMIT value. (The file structure is ruined.)
- A PUT call does not reposition the file.
- When the file can be shared (more than one instance of open could change the file at the same time), the task should take one of these actions:
  - Call LOCKK to lock the key value before it calls PUT.
  - Be prepared to process the $ERROR_STATUS value AA2075 returned if the key value is locked by another task.

PUTREP Call

Purpose

Replaces an existing record or writes a new record to a keyed file.

Format

CALL PUTREP (fit, working_storage_area, record_length, key_area, 0, 0, error_exit_procedure)

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

working_storage_area

Location from which data is copied.

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

---

record_length

Record length in bytes. The value is used only if the record type is variable length; it is ignored if the record type is fixed-length.

key_area

Variable containing the key value of the record to be written or replaced.

0

For CYBER 170 compatibility only. New programs should set this parameter to zero.

0

Reserved position for an unused parameter.

error_exit_procedure

Error-exit procedure name.

Remarks

- A PUTREP call requires at least append and shorten access as indicated by the $ACCESS_MODE or $ACCESS_AND_SHARE_ MODES value in the FIT. If alternate keys are defined in the file, a PUTREP call requires append, shorten, and modify access in order to update the alternate indexes.

- You always specify a primary-key value on a PUTREP call, not an alternate-key value, even if an alternate key is currently selected.

- The PUTREP call updates each alternate index that is to include the new record. If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the PUTREP call returns a nonfatal error.

- PUTREP executes a put request if the specified primary key does not match any existing primary key. It executes a replace request if a matching primary key is found in the file.

- If the file has variable-length (U or V) records, the length of the record written is the record_length value specified on the call. If omitted, the default is $WORKING_STORAGE_LENGTH value in the FIT.

  For a file with variable-length (U or V) records, the new record need not be the same length as the existing record; however, the new record length must be within the minimum and maximum record lengths for the file.

- If the file has fixed-length (F) records, the record_length value on the call is ignored. The fixed record length is always the length of each record written to the file.

  A warning message is issued for the first PUT, PUTREP, or REPLCE call whose record_length value differs from the fixed record length for the file. If the record_length is less than the fixed record length, the data is not padded so unknown data could be written at the end of the record. If the record_length is more than the fixed record length, the excess data is truncated.

- A PUTREP call does not reposition the file.

- Unlike a REPLC call, a PUTREP call does not require the task to own a Preserve_Access_and_Content or Exclusive_Access lock on the record.

  However, when the file is shared (more than one instance of open could exist), the task should either:

  - Call LOCKK to lock the key value before it calls PUTREP, or

  - Be prepared to process the abnormal status code AA2075 returned if the key value is locked by another task.

REPLC Call

Purpose Replaces an existing record in a keyed file.

Format **CALL REPLC (fit, working_storage_area, record_length, key_area, 0, 0, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

working_storage_area

Variable from which data is copied.

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

---

record_length

Record length in bytes. This parameter is used only if the record type is U or V; it is ignored if the record type is F.

key_area

Primary key value of the record to be replaced.

0

For CYBER 170 compatibility only. New programs should set this parameter to zero.

0

Reserved position for an unused parameter.

error_exit_procedure

Error-exit procedure name.

Remarks

- A REPLC call requires at least append and shorten access as indicated by the $ACCESS_MODE or $ACCESS_AND_SHARE_MODES value in the FIT. If alternate keys are defined in the file, a REPLC call requires append, shorten, and modify access in order to update the alternate indexes.
- If the file could be shared (more than one instance of open could change the file at the same time), a record can be replaced only by the owner of a Preserve_Access_and_Content or Exclusive_Access lock on the record.

  The task should lock the primary-key value by calling GET, GETN, or LOCKK before the REPLC call.

  To read about locks, see chapter 3, Sharing Keyed Files.
- A REPLC call always specifies a primary-key value, not an alternate-key value, even if an alternate key is currently selected.

- The new record must have the same primary-key value as the record being replaced. If REPLC cannot find a record with a matching primary-key value, it returns a nonfatal error.

- The REPLC call updates each alternate index that is to include the new record.

  If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the REPLC call returns a nonfatal error.

- A REPLC call does not reposition the file.

- If the record type for the file is variable (U or V), the record length is the $WORKING_STORAGE_LENGTH value in the FIT.

  For a file with variable (U or V) records, the new record need not be the same length as the existing record; however, the new record length must be within the minimum and maximum record lengths for the file.

- If the file has fixed-length (F) records, the record_length value on the call is ignored; the fixed record length for the file is always used.

  A warning message is issued for the first PUT, PUTREP, or REPLC call whose record_length value differs from the fixed record length for the file. If the record_length is less than the fixed record length for the file, the data is not padded so unknown data may be written at the end of the record. If the record_length is more than the fixed record length, excess data is truncated.

REWND Call

Purpose Rewinds the file.

Format CALL REWND (fit)

Parameters **fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

Remarks

- When the primary key is selected, REWND positions an indexed-sequential file at its lowest primary-key value and a direct-access file at the beginning of its first block.
- If the currently selected key is an alternate key, REWND positions the file to read the record with the lowest value for that alternate key.
- The $FILE_POSITION value after a successful REWND call is always 16 (end-of-record). It is not 1 (beginning-of-information).
- The file must be open when you issue the rewind request.

For Better Performance

Rewinding a file is more efficient than extensive backward skipping of records.

RMKDEF Call

Purpose Creates an alternate key in a keyed file.

NOTE

The NOS/VE RMKDEF call is provided for compatibility when migrating CYBER 170 programs that contain RMKDEF calls. When writing a new NOS/VE program, you should call the NOS/VE utility CREATE_ALTERNATE_INDEXES using the SCLCMD call. The CREATE_ALTERNATE_INDEXES utility is described in the NOS/VE Advanced File Management Usage manual.

RSBUILD Call

Purpose Gets primary-key values from a keyed file and combines them with a result set.

Format **CALL RSBUILD (fit, source_result_set, target_result_set, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, logical_operation, new_result_placement, actual_result_set_placement, condition_code)**

Parameters

fit

Name of the variable containing the FIT pointer for the keyed file.

source_result_set

Identifier of the result set to be combined (as returned by its RSOPEN call).

target_result_set

Identifier of the target result set (as returned by its RSOPEN call).

low_key

Key value at which the range begins. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the low_key value.

low_key_relation

Indicates where the range begins in relation to the lowest key value in the range. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Exclude the lowest key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Include the lowest key value.

'LOWEST_KEY' or 'LK'

Start at the beginning of the index. (Ignore the low_key and major_low_key values.)

high_key

Key value at which the range ends. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

If the high_key value is less than the low_key value, RSBUILD returns a nonfatal condition code value.

major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the high_key value.

high_key_relation

Indicates where the range begins in relation to the highest key value in the range. Options are:

'GREATER_KEY' or 'GK' or 'GT'

Include the highest key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Exclude the highest key value.

'HIGHEST_KEY' or 'HK'

Stop at the end of the index. (Ignore the high_key and major_high_key values.)

logical_operation

Integer specifying the logical operation performed to combine the source result set with the new range of key values. Options are:

0 Logical AND. The combined result set is the intersection of the original result sets. It contains only those key values that belong to both of the original sets.

1 Logical OR. The combined result set is the union of the original result sets. It contains all key values from both original result sets.

2 Logical XOR. The combined result set is the union of the original result sets without the intersection of the original result sets. It contains all key values from each of the original result sets that do not belong also to the other result set.

new_result_placement

Integer specifying the result set file to which the combined result set is written. Options are:

0 The combined result set overwrites the source result set. Use this value when the source result set is no longer needed.

1 The combined result set is written to the target result set. Use this value when the source_result_set is to be saved for later use. It is also used on the initial AMP$BUILD_RESULT_SET call for a new result set.

2 The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the variable specified by the actual_result_set_placement parameter on the call. Use this value when the source result set is no longer needed and the source and target result sets differ.

actual_result_set_placement

Integer variable in which the call indicates the result set file to which the combined result set has been written. Options are:

0 The source result set has been overwritten.

1 The combined result set has been written to the target result set; the source result set has been preserved.

condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion. For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- RSBUILD adds a range of key values to a result set. It can be used to:

  - Add primary-key values to an empty result set.

    For this use, the call specifies the same result set as the source_result_set and as the target_result_set, but the new_result_placement value should be 1 (result_in_target). The logical_operation value should be 1 (logical OR).

  - Add primary-key values to an existing result set. The combined result set can overwrite the source result set or be written to the target result set.

    When the source result set is to be overwritten, the call specifies the same result set as the source_result_set and as the target_result_set. The new_result_placement value should be 0 (result_in_source).

    When the source result set is to be kept, the call specifies different result sets as the source result set and as the target result set and the new result placement value 1 (result in target).

  - When two result sets are specified, but it does not matter whether the source_result_set is overwritten, specify the new_result_placement value 2 (result_in_fastest_place).

- The specified data file, source result set, and target result set must be open. The data file is opened by an OPENM call; the result set is opened by an RSOPEN call.

  The data file and nested file identification in the result set files must match the data file cycle opened using the FIT and the nested file specified in the FIT. The file identification is stored in the result set when the result set is first opened.

The currently selected nested file for the data file must be the nested file specified on the RSOPEN call. The nested file selected when the file is opened is the default nested file, $MAIN_FILE. To select another nested file, store the nested file name as the $NESTED_FILE_NAME value in the FIT.

- The currently selected key must be the key whose index is to be searched for the range specified on the call. The key selected when the file is opened is the primary key ($PRIMARY_KEY); to select another key, call STOREF to store the key name in the FIT.

  NOTE

  For a direct-access file, the selected key must be an alternate key. RSBUILD cannot use the primary key of a direct-access file.

- The search for the range specified on the call is the same as the range search performed by KLCOUNT. For more information, see the KLCOUNT call description.

- After finding the specified range in the index, the call gets the primary-key values from the index. If the index is for an alternate key which allows duplicate values, the call gets the list of primary-key values for each alternate-key value in the range.

- RSBUILD cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

RSCLEAR Call

Purpose Discards the existing result set in the result set file.

Format **Call RSCLEAR (result_set_id, condition_code)**

Parameters

result_set_id

Identifier of the result set to be cleared (as returned by its RSOPEN call).

condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on deciphering the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks Use the RSCLEAR call to erase the existing result set in a result set file after it has been opened by an RSOPEN call. After the file is cleared, it is equivalent to a new result set file.

RSCLOSE Call

Purpose

Closes an open result set.

Format

Call RSCLOSE (result_set_id, condition_code)

Parameters

result_set_id

Result set identifier (as returned by its RSOPEN call).

condition_code

Integer variable in which the error status value is returned. Return of a zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- Closing a result set prevents further operations using the result set until it is opened again.
- If an RSCLOSE call is not issued for an open result set, the result set is closed at task termination.
- A closed result set continues to exist until its file is deleted.

RSCOMB Call

Purpose Combines two result sets.

Format **CALL RSCOMB (first_result_set, second_result_set, target_result_set, logical_operation, new_result_placement, actual_result_set_placement, condition_code)**

Parameters

first_result_set

Identifier of the first result set to be combined (as returned by its RSOPEN call).

second_result_set

Identifier of the second result set be combined (as returned by its RSOPEN call).

If the new_result_placement parameter specifies 0 (result in source), the second source_result_set is overwritten.

target_result_set

Identifier of the target result set (as returned by its RSOPEN).

logical_operation

Integer specifying the logical operation performed to combine the two source result sets.

0 Logical AND. The combined result set is the intersection of the original result sets. It contains only those key values that belong to both of the original sets.

1 Logical OR. The combined result set is the union of the original result sets. It contains all key values from both original result sets.

2 Logical XOR. The combined result set is the union of the original result sets without the intersection of the original result sets. It contains all key values from each of the original result sets that are not in both of the original result sets.

new_result_placement

Integer specifying the result set file to which the combined result set is written.

0 The combined result set overwrites the second source result set. Use this value only when the second source result set is no longer needed or the second source result set and the target result set are the same.

1 The combined result set is written to the target result set. Use this value when the second source result set is to be saved for later use.

2 The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the actual_result_set_placement variable. Use this value when the second source result set is no longer needed and the second source result set and target result set differ.

actual_result_set_placement

Integer variable in which the call indicates the result set file to which the combined result set has been written.

0 Result in source. The second source result set has been overwritten.

1 Result in target. The combined result set has been written to the target result set file; the second source result set has been preserved.

condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_ STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- The RSCOMB call performs the same combination operations that can be performed by an RSBUILD call. When possible, use RSBUILD to perform the combination at the same time the key values are taken from the data file.
- All result sets specified on the call must be open. However, the data file to which the result set applies need not be open.

  If the data file is open, its selected nested file need not be the nested file to which the result set applies. This is because the RSCOMB call does not require any information from the data file.
- All result sets specified on the call must apply to the same keyed file cycle and nested file in the keyed file. (The first RSOPEN call for a result set stores the identification of the data file cycle and nested file to which the result set file applies in the result set.)
- RSCOMB cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

RSDLTE Call

Purpose

Deletes a primary-key value from a result set.

Format

CALL RSDLTE (target_result_set, key_location, condition_code)

Parameters

target_result_set

Identifier of the result set from which the primary-key value is deleted (as returned by its RSOPEN call).

key_location

Location of the primary-key value to be deleted from the result set.

condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- If the key value is not in the result set, the call does nothing and no message is issued.
- Use this call when only a few scattered primary-key values need to be deleted from the result set.

  When several primary-key values need to be deleted, it is more efficient to create a temporary result set containing those values and combine it with the original result set.

  For more information, see Adding and Deleting Key Values in the Result Sets description in chapter 5, Result Sets.
- This call can only specify a primary-key value. It cannot specify an alternate-key value.

  However, you can delete the primary-key values associated with an alternate-key value from the result set. To do so, perform the following steps:

  1. Select the alternate key.
  2. Call RSBUILD specifying the logical XOR (2) operation to remove the key values. It specifies the key values to be removed as a range containing only the one alternate-key value. (The low_key and high_key values of the range are the same.)
  3. If any of the primary-key values in the alternate key list might not be in the original result set, combine the target result set again with the original result set using a logical AND (0) operation.
- RSDLTE cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

RSGETN Call

Purpose Reads a record from a keyed file using a result set.

Format **CALL RSGETN (fit, source_result_set, result_set_not, working_storage_area, key_area, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

source_result_set

Identifier of the result set used to read the record (as returned by its RSOPEN call).

result_set_not

Indicates whether the call reads the next record that is in the result set or the next record that is not in the result set.

'NO'

Reads the next record in the result set.

'YES'

Reads the next record NOT in the result set.

working_storage_area

Location to which the record data is copied. The default is the $WORKING_STORAGE_AREA value in the FIT.

key_area

Variable in which the primary-key value of the record is returned. (The default is the $KEY_AREA value in the FIT.

error_exit_procedure

Error exit procedure name. The default is the $ERROR_EXIT_PROCEDURE_NAME value in the FIT.

Remarks

- Use RSGETN to read a sequence of records. The sequence of records can be the records in the result set or all records in the data file that are not in the result set.

  To read the records in the result set, specify 'NO' for the result_set_not parameter on each RSGETN call. To read the records NOT in the result set, specify 'YES' for the result_set_not parameter on each RSGETN call.

  NOTE

  For a direct-access file, RSGETN can read the sequence of records in the result set, but it cannot read the sequence of records not in the result set. In other words, the NOT operator is invalid so each RSGETN call for a direct-access file must specify 'NO' for the result_set_not parameter.

- The data file must be open. The selected nested file must be the nested file specified when the result set was first opened. The selected key for the nested file must be its primary key.

- The RSOPEN call establishes the result set position at its beginning. (The result set is also positioned at its beginning by any of the calls that change the result set.)

  Each RSGETN call without the NOT operator repositions the result set forward one primary-key value. RSGETN calls with the NOT operator position the result set forward as needed.

  RSREWND, RSSTART, and RSSKIP calls explicitly reposition the result set.

- Other calls can intervene between RSGETN calls. However, calls that reposition the data file must not intervene between RSGETN calls.

- An RSGETN call without the NOT operator issues a GET call using the primary-key value at the current result set position. It then advances the result-set position one primary-key value.

- RSGETN calls with the NOT operator get the records that are not in the result set. The first get_not call establishes the starting data file position.

  It does so by reading the primary-key value at the current result set position and then positioning the data file at that record. The first get_not call then reads a record, the same as subsequent get_not calls, as follows:

- Each get_not call performs the following steps:

  1. Calls GETN to read the record at the current data file position. (GETN reads a record and advances the data file position one record.)

2. Compares the primary-key value returned by the GETN call and the primary-key value at the current position of the result set.

   a. If the values match, it:

      Discards the record read, advances the result set position forward one value, and continues at step 1.

   b. If the values do not match, it:

      Terminates, leaving the record read in the working storage area.

- Each call that returns a record, including the last record in the sequence, returns the $FILE_POSITION value 16 in the FIT. The next RSGETN call after the call that returns the last record returns the value 64, indicating that the end of the sequence has been reached. The call that returns 64 copies no data to the working storage area.

  A $FILE_POSITION value of 64 returned by a get call indicates that the result set is positioned at its end, and so all records in the result set have been read.

  A $FILE_POSITION value of 64 returned by a get_not call, indicates that the data file, as well as the result set, is positioned at its end, and so all records not in the result set have been read.

RSINFO Call

Purpose Returns current information about a result set.

Format **Call RSINFO (result_set_id, previous_key, next_key, key_count, keys_remaining, position, condition_code)**

Parameters

result_set_id

Identifier of the result set for which information is returned (as returned by its RSOPEN call).

previous_key

Variable in which the call returns the preceding primary-key value in the result set. Use an integer variable for an integer primary key; use a character variable for a collated or uncollated primary key. (A previous_key value is returned only when the position returned is 1 or 2.)

next_key

Variable in which the call returns the next primary-key value in the result set. Use an integer variable for an integer primary key; use a character variable for a collated or uncollated primary key. (A next_key value is returned only when the position returned is 0 or 1.)

key_count

Integer variable in which the call returns the number of primary-key values in the result set.

keys_remaining

Integer variable in which the call returns the number of primary-key values from the current position to the end of the result set.

position

Integer variable in which the call returns the relative position of the result set. Options are:

0 Positioned at its beginning (previous_key undefined).

1 Positioned somewhere between its beginning and end. (Both the previous_key and the next_key values are defined.)

2 Positioned at its end (next_key undefined).

3 The result set is empty. (The previous_key and next_key values are both undefined.)

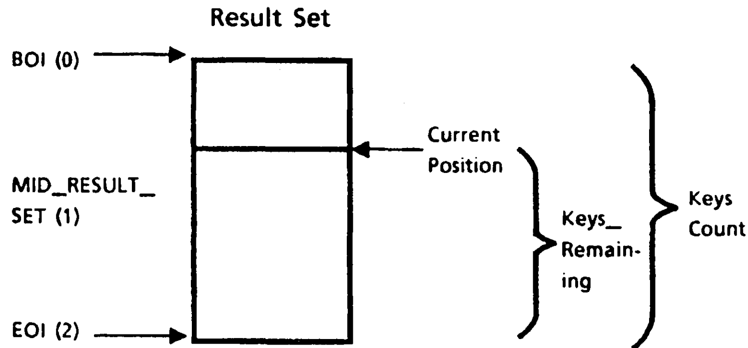
condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- The following figure illustrates the count and position values returned.



- Assuming the result set has not been repositioned since the call, the previous_key value is the primary_key value used by the last RSGETN call and the next_key value is the primary-key value that will be used by the next RSGETN call.

- The key_count value returned is the number of primary-key values in the result set and therefore, the number of records that a series of get calls could fetch using the result set.

  However, to determine the number of records that series of get_not calls could fetch, your program must know the total number of records in the nested file and subtract the key_count value from that number. The record count for a nested file is not available from the keyed-file interface although you can display it using the DISPLAY_KEYED_FILE_PROPERTIES command.

- RSINFO calls do not change the result set position or the data file position and so do not disrupt a sequence of RSGETN calls.

RSOPEN Call

Purpose Opens a result set and positions it at its beginning.

Format **CALL RSOPEN (result_set_file, data_file, nested_file, result_set_id, condition_code)**

Parameters

result_set_file

File name of the result set file to be opened. The result set file name is a 1- to 256-character string. If you do not specify a file path, the file is assumed to be in the working catalog. If an existing result set file is to be used, it must be attached with at least read access. Otherwise, if the result set file does not exist, RSOPEN creates it.

data_file

File name of the keyed file to which the result set applies. The file name is a 1- to 31-character string. If you do not specify a file path, the file is assumed to be in the working catalog. The file must be attached with at least read access.

nested_file

Name of a nested file in the data file. The nested file name is a 1- to 31-character string. To specify the default nested file, specify either the name $MAIN_FILE or all blanks.

result_set_id

Integer variable in which the result set identifier is returned. It is used by later result set calls to identify the open result set.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- A result set must be opened by an RSOPEN call before it can be used for any purpose. The result set remains open until it is closed by an RSCLOSE call or by the termination of the task.

- If the specified result set file does not exist or is not attached, RSOPEN creates a new temporary file and records in its file attributes that it is a result set. It also stores the identification of the specified data file and nested file in the result set.

- If the specified result set file is in the $LOCAL catalog, RSOPEN checks its attributes to ensure that it is a result set file. It also checks that the data file and nested-file identification in the result set file matches that of the data file and nested file specified on the call.

- The RSOPEN returns the identifier that all subsequent result set calls use to specify the result set.

NOTE

Do not change the contents of the result_set_id variable while the result set is open; any change would invalidate the identifier.

- The same result set identifier can be used with different instances of open of the data file. However, the result set may no longer be correct after the data file is updated. For more information, see Result Set Validity in chapter 5, Result Sets.
- A result set file can have only one instance of open at a time.

RSPUT Call

Purpose Adds a primary-key value to the result set.

Format **CALL RSPUT (target_result_set, key_location, condition_code)**

Parameters

target_result_set

Identifier of the result set to which the primary-key value is added (as returned by its RSOPEN call).

key_location

Location of the primary-key value to be added to the result set.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- Use RSPUT to perform either of these actions:
  - Directly add a few scattered primary-key values to the result set.
  - Create a temporary result set of scattered primary-key values to be added or deleted from the result set.
- When several primary-key values need to be added, it is more efficient to create a temporary result set containing those values and combine it with the original result set. For more information, see Adding and Deleting Key Values in the Result Sets description in chapter 5, Result Sets.

  For Better Performance

  If possible, put primary-key values into a result set in ascending order.

- This call can specify a primary-key value only. It cannot specify an alternate-key value.

  However, you can add the primary-key values associated with an alternate-key value to the result set. To do so, perform the following steps:

  1. Select the alternate key.
  2. Call RSBUILD specifying the logical OR (1) operation to add the key values. The specified key-value range should contain only the one alternate-key value. (The low_key and high_key values of the range are the same value.)

- RSPUT cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

RSREWND Call

Purpose Repositions a result set at its beginning.

Format **CALL RSREWND (source_result_set, condition_code)**

Parameters

source_result_set

Identifier of the result set to be rewound (as returned by its RSOPEN call).

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks The result set is also positioned at its beginning by an RSOPEN call and by any result set call that changes the result set.

RSSKIP Call

Purpose Repositions a result set forward or backward.

Format CALL RSSKIP (source_result_set, count, condition_code)

Parameters

source_result_set

Identifier of the result set to be repositioned (as returned by its RSOPEN call).

count

Number of primary-key values to be skipped. A positive integer causes a skip forward (toward the end of the result set); a negative integer causes a skip backward (toward the beginning of the result set).

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- A skip forward that encounters the end of the result set does not return an error. The result set is left positioned at its end. The next RSGETN call returns no data and a $FILE_POSITION value of 64 in the FIT.
- Similarly, a skip backward that encounters the beginning of the result set does not return an error. The result set is left positioned at its beginning.
- If necessary, the program can call RSINFO to get the result set position after a skip.

RSSTART Call

Purpose

Positions a result set using a primary-key value.

Format

CALL RSSTART (source_result_set, key_location, major_key_length, key_relation, condition_code)

Parameters

source_result_set

Identifier of the result set to be repositioned (as returned by its RSOPEN call).

key_location

Location containing the primary-key value at which the result set is to be positioned.

major_key_length

Indicates whether the primary-key value is to be located by major key. A zero value specifies that a major key is not used; a nonzero value specifies the number of bytes in the major key.

key_relation

Indicates whether the primary-key value in the file must match the primary-key value specified on the call.

0 The primary-key values must match.

1 If a matching primary-key value is not found, the next greater primary-key value is used.

2 The first primary-key value found that is greater than the specified primary-key value is used.

condition_code

Integer variable in which the condition code is returned. A zero value indicates successful completion.

For information on translating the condition_code, see the $ERROR_STATUS description in chapter 7, File Information Table Keywords and Values.

Remarks

- The RSSTART call establishes the result set position at the primary-key value specified by the call. Subsequent get or get_not calls use only the result set values from the start position to the end of the result set.

SKIP Call

Purpose Repositions a keyed file either forward or backward the specified number of records.

Format **CALL SKIP (fit, count)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

count

Number of records to be skipped. A positive integer causes a skip forward (toward the end-of-information); a negative integer causes a skip backward (toward the beginning-of-information).

If zero is specified for the count parameter, the $SKIP_COUNT value in the FIT is used. If it is also 0, no skipping is done.

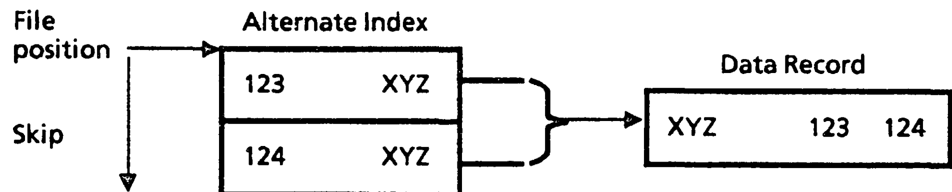
Remarks

- A SKIP call requires read access to the file.
- A SKIP call skips records in order by key value. This is because it actually skips key values in the index for the key.

  SKIP calls are valid only when an index exists for the selected key. Thus, SKIP calls are not valid while the primary key of a direct-access file is selected.

  If the currently selected key is an alternate key, it skips records in order by alternate-key value.

  The same record may be skipped more than once if it contains more than one alternate-key value. For example, suppose a record with primary-key value XYZ contains two integer alternate-key values, 123 and 124. Assume that the file is positioned to read the record with alternate-key value 123 as follows:



  A SKIP call to skip one record forward skips forward one alternate-key value in the alternate index. The file is then positioned to read the data record for alternate-key value 124, which is also the data record for alternate-key value 123.

For Better Performance

A skip call should be used for skipping a few records only, because each intervening record is read and counted, which increases execution time. A random read request takes less time than a lengthy skip request.

- The $FILE_POSITION value after a SKIP call is always 16, end-of-record unless:
  - The SKIP reaches a file boundary. Then the $FILE_POSITION value is 1, beginning-of-information, or 64, end-of-information.
  - An alternate key is selected. Then the $FILE_POSITION value is 8, end-of-key-list.
- A SKIP reaches a file boundary only when it cannot skip the requested number of records in the requested direction.

  For example, suppose the primary key is selected and the file is positioned to read the third record (the $FILE_POSITION is 16):

  BOI..record1..record2..record3..EOI

  If a SKIP skips backward two records, the SKIP does not reach a file boundary and the $FILE_POSITION value is still 16: If the SKIP skips backward another record, it reaches the file boundary and the

  BOI..record1..record2..record3..EOI

  $FILE_POSITION value is 1. (The first record can be read from this position or the preceding position.)

  BOI..record1..record2..record3..EOI

  A SKIP now skips forward three records and the $FILE_POSITION value is 16.

  BOI..record1..record2..record3..EOI

  A read at this position or one more skip forward reaches the file boundary, and the $FILE_POSITION value is 64.

  BOI..record1..record2..record3..EOI

  A skip backward one record positions the file to read the last record and the $FILE_POSITION value is 16.

  BOI..record1..record2..record3..EOI

- When a skip encounters the end-of-information or the beginning-of-information, it returns a nonfatal error.

  In either case, SKIP calls the DX procedure if one is specified in the FIT.

  If the program immediately calls SKIP again to skip in the same direction, SKIP calls the error-exit procedure (if one is specified in the FIT).

- If the skip reaches a file boundary and cannot be completed, the $SKIP_COUNT value in the FIT is the number of records that could not be skipped. The number of records actually skipped can be calculated by subtracting the residual skip count from the requested skip count.

STARTM Call

Purpose Positions a keyed file using the specified key value and key relation.

Format **CALL STARTM (fit, key_area, 0, major_ley_length, error_exit_procedure)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

key_area

Key value used to position the file.

0

For CYBER 170 compatibility only. New programs should set this parameter to zero.

major_key_length

For a fixed-length key, it is the length of the major key in bytes. A zero value indicates that the full key value is used.

For a variable-length key, a nonzero value is required because it specifies the length, in bytes, of the specified key value.

If the major_key_length is zero, the $MAJOR_KEY_LENGTH value in the FIT is used. However, the $MAJOR_KEY_LENGTH value is always reset to zero after any call with an major_key_length parameter.

error_exit_procedure

Error-exit procedure name.

Remarks

- A STARTM call requires read access to the file.
- STARTM searches for the key value in the index of the selected key in the selected nested file only.

  A STARTM call is valid only when an index exists for the selected key. Thus, STARTM calls are invalid while the primary key of a direct-access file is selected.
- If an alternate key has been selected and the key is a concatenated key, the values for the pieces of the concatenated key are assembled in the key area. The pieces must be concatenated in the key area in the order defined for the alternate key.

  For example, if the key is the last 3 bytes of the record followed by the first 3 bytes of the record, the value in the key area must be the value of last 3 bytes followed by the value of the first 3 bytes.

- STARTM searches for the first key value that satisfies the relation specified by the $KEY_RELATION value in the FIT.
  - If the relation is EQUAL_KEY and an equal key value does not exist in the file, STARTM returns a nonfatal error. The file is left positioned to read the next record (the record that would follow the specified record if it existed).
  - If the $KEY_RELATION value is GREATER_OR_EQUAL_KEY or GREATER_KEY and no key value in the file satisfies the relation, the data-exit (DX) procedure is called, if one is specified in the FIT. The file is left positioned at the end-of-information.
- STARTM cannot return a file position of 1 (beginning-of-information). When the key value to be found is less than any key value in the file, STARTM returns a file position value of 8 or 16 (end-of-key-list or end-of-record).
- If the major_key_length is 0, the $MAJOR_KEY_LENGTH value in the FIT is used. The $MAJOR_KEY_LENGTH value in the FIT is cleared after any call having a major_key_length parameter. For more information, see the $MAJOR_KEY_LENGTH FIT value description in chapter 7, File Information Table Keywords and Values.

  A nonzero major_key_length value is required while a variable-length alternate key is selected. Otherwise, a nonzero value is specified only when a major-key search is to be used.
- A STARTM call does not return a record to the working storage area.
- When an alternate key is selected and a primary-key area is specified in a FIT, a STARTM call returns the primary-key value of the record at which the file is positioned. The value is returned in the primary-key area.

STOREF Call

Purpose Stores a value in the FIT, which applies to the file the next time the file is opened.

Format **CALL STOREF (fit, fit_keyword, fit_value)**

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

fit_keyword

Character expression specifying a FIT keyword. The keyword can be specified using uppercase and/or lowercase letters. For more information on FIT keywords, see chapter 7, File Information Table Keywords and Values.

fit_value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description in chapter 7, File Information Table Keywords and Values. Character values can be specified using uppercase and/or lowercase letters.

Remarks

- You can call STOREF any time after the FILEIS or FILEDA call that returns FIT pointer.
- To clear a FIT value, specify the keyword and a 0 on a STOREF call.

  For example, suppose you previously specified a primary-key area, but now no longer want any primary-key values returned. To prevent this, you clear the $PRIMARY_KEY_ADDRESS value as follows:

  ```
  CALL STOREF(fit, '$PRIMARY_KEY_ADDRESS', 0)
  ```

- Preserved file attribute values cannot be changed after the file has been first opened. These include:

$EMBEDDED_KEY	$MAXIMUM_BLOCK_LENGTH
$FILE_ORGANIZATION	$MAXIMUM_RECORD_LENGTH
$KEY_LENGTH	$MINIMUM_RECORD_LENGTH
$KEY_POSITION	$RECORD_TYPE
$KEY_TYPE	

  Specifying a value for $KEY_LENGTH or $KEY_POSITION after the file is first opened does not change the preserved attributes (the primary key length and position). Instead, the $KEY_LENGTH and $KEY_POSITION values can be used to select an alternate key or to specify the sparse-key control position for an RMKDEF call.

Examples

- This call specifies that the key value is to be returned in the variable RETKEY. (RETKEY should be in a common block.)

```
CALL STOREF(fit, '$KEY_ADDRESS', retkey)
```

- This call specifies the primary-key starting position as the beginning of the record.

```
CALL STOREF(fit, '$KEY_POSITION', 0)
```

- This call clears the error-exit procedure specification.

```
CALL STOREF(fit, '$ERROR_EXIT_PROCEDURE', 0)
```

UNLOCKF Call

Purpose Clears a file lock for the currently selected nested file.

Format CALL UNLOCKF (fit)

Parameters **fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

Remarks

- An UNLOCKF call clears only the file lock for the nested file specified by the $NESTED_FILE_NAME value in the FIT. It clears only the lock belonging to the instance of open.

  An UNLOCKF call clears only one nested file lock. It does not clear any other file locks or any key-value locks. To clear individual key-value locks or all locks, use UNLOCKK.

- When a lock expires, the task must clear the lock before it can perform any more operations on the instance of open. To clear all locks belonging to the instance of open (both file and key locks), call UNLOCKK with the 'ALL' parameter value specified.

- To read about lock expiration, see Lock Expiration and Clearing in chapter 3, Sharing Keyed Files.

- UNLOCKF cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

UNLOCKK Call

Purpose

Clears either a single key-value lock or all locks for the currently selected nested file.

Format

CALL UNLOCKK (fit, key_area, 'ALL')

Parameters

fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

key_area

Location containing the primary-key value to be unlocked. Specify 0 for this parameter if 'ALL' is specified.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

'ALL'

Requests clearing of all locks for this instance of open. If you specify 'ALL' as the third parameter value, specify 0 as the key_area parameter.

Remarks

- An UNLOCKK call performs one of two operations depending on whether the third parameter value ('ALL') is specified:
  - If 'ALL' is specified, UNLOCKK clears all locks for the currently selected nested file (both the file lock, if any, and all key-value locks, if any).
  - If 'ALL' is omitted, UNLOCKK clears only the lock for the primary-key value at the specified key location (key_area).
- A key value lock can be cleared without an UNLOCKK call:
  - It is cleared when the instance of open is closed.
  - If automatic unlock was requested for the lock, it is cleared when the task issues another call for the instance of open (other than an IFETCH or STOREF). (The lock is unlocked even if the request fails.)

  NOTE

  Do not call UNLOCKK to clear a key-value lock requested with automatic unlock. Such a call would first perform the automatic unlock and then the UNLOCKK operation. The second unlock operation would find no lock on the key value and issue a nonfatal error.

- When 'ALL' is specified and no locks exist for the nested file, no error is returned. However, if 'ALL' is omitted and the instance of open does not own a lock on the key value, UNLOCKK returns a nonfatal error.

- When a lock expires, the task must clear the expired lock before it can perform any more operations on the instance of open.

  The task is notified that a lock has expired by the status returned by the next operation attempted. However, it is not notified as to which lock has expired.

  When notified that an expired lock exists, the task can either clear all locks or clear each lock individually. It can clear all locks by calling UNLOCKK with the 'ALL' option. An UNLOCKF call clears an individual file lock; UNLOCKK calls can clear individual key locks.

- To read about lock expiration, see Lock Expiration and Clearing in chapter 3, Sharing Keyed Files.

- UNLOCKK cannot be called for an instance of open for which a parcel is in progress. For a description of parcels, see chapter 4, Parcels.

Examples

- This call clears the lock on the key value in the variable specified by the $KEY_ADDRESS value in the FIT:

```
CALL UNLOCKK (fit)
```

- This call clears the lock on the key value in variable KEY1 (and stores KEY1 as the $KEY_ADDRESS in the FIT):

```
CALL UNLOCKK (fit, key1)
```

- This call clears all key-value and file locks for the currently selected nested file.

```
CALL UNLOCKK (fit, 0, 'ALL')
```

File Information Tables 7

How to Use FIT Keywords in FORTRAN Programs 7-3
FIT Keywords 7-3
Abbreviations for FIT Keywords and Values 7-3

FIT Keywords and Values: Quick Reference 7-4
$ACCESS_AND_SHARE_MODES or $AASM 7-5
$ACCESS_MODE or $AM 7-8
$AUTOMATIC_UNLOCK or $AU 7-11
$AVERAGE_RECORD_LENGTH or $ARL 7-12
$COLLATE_TABLE or $CT 7-13
$COLLATE_TABLE_NAME or $CTN 7-14
$COMPRESSION_PROCEDURE_NAME or $CPN 7-15
$DATA_PADDING or $DP 7-17
$DELETE_DATA or $DD 7-18
DX (Data Exit Procedure) 7-19
$EMBEDDED_KEY or $EK 7-20
$ERROR_COUNT or $EC 7-21
$ERROR_EXIT_PROCEDURE_NAME or $ERROR_EXIT_NAME or $EEPN or $EEN 7-22
$ERROR_EXIT_PROCEDURE or $EEP 7-23
$ERROR_LIMIT or $EL 7-24
$ERROR_STATUS or $ES 7-25
$ESTIMATED_RECORD_COUNT or $ERC 7-27
$FILE_IDENTIFIER or $FI 7-28
$FILE_ORGANIZATION or $FO 7-29
$FILE_POSITION or $FP 7-30
$FORCED_WRITE or $FW 7-31
FNF (Fatal/Nonfatal Flag) 7-32
$GET_AND_LOCK or $GAL 7-33
$GLOBAL_ACCESS_MODES or $GAM 7-34
$GLOBAL_SHARE_MODES or $GSM 7-35
$HASHING_PROCEDURE_NAME or $HPN 7-36
$INDEX_LEVELS or $INDEX_LEVEL or $IL 7-37
$INDEX_PADDING or $IP 7-38
$INITIAL_HOME_BLOCK_COUNT or $IHBC 7-39
$KEY_ADDRESS or $KA 7-40
$KEY_LENGTH or $KL 7-41
$KEY_NAME or $KN 7-42
$KEY_POSITION or $KP 7-43
$KEY_RELATION or $KR 7-44
$KEY_TYPE or $KT 7-45
$LAST_OPERATION or $LO 7-46
$LOCAL_FILE_NAME or $LFN 7-47
$LOCK_EXPIRATION_TIME or $LET 7-48
$LOCK_INTENT or $LI 7-49
$LOG_RESIDENCE or $LR 7-50
$LOGGING_OPTIONS 7-51
$MAJOR_KEY_LENGTH or $MKL 7-52
$MAXIMUM_BLOCK_LENGTH or $MAXBL 7-53
$MAXIMUM_RECORD_LENGTH or $MAXRL 7-54
$MESSAGE_CONTROL or $MC 7-55
$MINIMUM_RECORD_LENGTH or $MINRL 7-56
$NESTED_FILE_NAME or $NFN 7-57

OC (Open/Close Flag) 7-58
ON (Old/New Flag) 7-59
$OPEN_POSITION or $OP 7-60
$OPEN_SHARE_MODES or $OSM 7-61
$PASSWORD or $P 7-63
$PRIMARY_KEY_ADDRESS or $PKA 7-64
RL (Record Length) 7-65
$RECORD_LIMIT or $RL 7-66
$RECORD_TYPE or $RT 7-67
$RECORDS_PER_BLOCK or $RPB 7-68
$SKIP_COUNT or $SC 7-69
$WAIT_FOR_ATTACHMENT or $WFA 7-70
$WAIT_FOR_LOCK or $WFL 7-71
$WORKING_STORAGE_ADDRESS or $WSA 7-72
$WORKING_STORAGE_LENGTH or $WSL 7-73

File Information Tables 7

These are the keywords used to store and fetch values from a File Information Table (FIT). You specify FIT keywords and values on the keyed-file interface calls FILEIS, FILEDA, IFETCH, STOREF, and OPENM.

Most FIT keywords refer to file attributes. Other FIT keywords refer to information used only by the FORTRAN keyed-file interface.

FIT Keyword	Abbreviation
$ACCESS_AND_SHARE_MODES	$AASM
$ACCESS_MODE	$AM
$AUTOMATIC_UNLOCK	$AU
$AVERAGE_RECORD_LENGTH	$ARL
$COLLATE_TABLE	$CT
$COLLATE_TABLE_NAME	$CTN
$COMPRESSION_PROCEDURE_NAME	$CPN
$DATA_PADDING	$DP
$DELETE_DATA	$DD
DX	DX
$EMBEDDED_KEY	$EK
$ERROR_COUNT	$EC
$ERROR_EXIT_PROCEDURE_NAME	$EEPN
$ERROR_EXIT_PROCEDURE	$EEP
$ERROR_LIMIT	$EL
$ERROR_STATUS	$ES
$ESTIMATED_RECORD_COUNT	$ERC
$FILE_IDENTIFIER	$FI
$FILE_ORGANIZATION	$FO
$FILE_POSITION	$FP
FNF	FNF
$FORCED_WRITE	$FW
$GET_AND_LOCK	$GAL
$GLOBAL_ACCESS_MODES	$GAM
$GLOBAL_SHARE_MODES	$GSM
$HASHING_PROCEDURE_NAME	$HPN
$INDEX_LEVELS	$IL
$INDEX_PADDING	$IP
$INITIAL_HOME_BLOCK_COUNT	$IHBC
$KEY_ADDRESS	$KA

FIT Keyword	Abbreviation
$KEY_LENGTH	$KL
$KEY_NAME	$KN
$KEY_POSITION	$KP
$KEY_RELATION	$KR
$KEY_TYPE	$KT
$LAST_OPERATION	$LO
$LOCAL_FILE_NAME	$LFN
$LOCK_EXPIRATION_TIME	$LET
$LOCK_INTENT	$LI
$LOG_RESIDENCE	$LR
$LOGGING_OPTIONS	(No abbreviation)
$MAJOR_KEY_LENGTH	$MKL
$MAXIMUM_BLOCK_LENGTH	$MAXBL
$MAXIMUM_RECORD_LENGTH	$MAXRL
$MESSAGE_CONTROL	$MC
$MINIMUM_RECORD_LENGTH	$MINRL
$NESTED_FILE_NAME	$NFN
OC	OC
ON	ON
$OPEN_POSITION	$OP
$OPEN_SHARE_MODES	$OSM
$PASSWORD	$P
$PRIMARY_KEY_ADDRESS	$PKA
$RECORD_LIMIT	$RL
$RECORD_TYPE	$RT
$RECORDS_PER_BLOCK	$RPB
RL	RL
$SKIP_COUNT	$SC
$WAIT_FOR_ATTACHMENT	$WFA
$WAIT_FOR_LOCK	$WFL
$WORKING_STORAGE_ADDRESS	$WSA
$WORKING_STORAGE_LENGTH	$WSL

How to Use FIT Keywords in FORTRAN Programs

Unless indicated otherwise in the following section, a FIT value specified on a keyed-file interface call is stored in the FIT.

The values for a FIT can be set by four different methods. When a keyed file is opened and a FIT is created, the values for the FIT are set by one of these methods. The methods, in their order of precedence, are:

1. Attribute values specified on a NOS/VE SET_FILE_ATTRIBUTES command.
2. For existing files, the attribute values stored with the file.
3. The attribute values specified by the FILEIS or FILEDA call or a STOREF call before the OPENM call.
4. The default values for the attributes.

NOS/VE file attributes fall into three categories: returned attributes, temporary attributes, and preserved attributes.

- Returned attributes are attributes whose values can be fetched but cannot be specified.
- Temporary attributes are attributes that are not stored with the file and may be changed each time the file is opened.
- Preserved attributes are attributes that are stored with the file when it is first opened and are preserved for the lifetime of the file.

In general, you cannot change a preserved file attribute after the file has been first opened. However, you can specify a preserved file attribute value in the FIT of an existing file to verify that the correct file is being used. For example, if you set the $FILE_ORGANIZATION value to indexed-sequential in the FIT, the OPENM call checks that the preserved $FILE_ORGANIZATION attribute is indexed-sequential. If it is not, OPENM returns an error.

FIT Keywords

To specify or fetch a FIT value, you specify the keyword for that value.

You can specify FIT keywords and character values using uppercase and/or lowercase letters.

Abbreviations for FIT Keywords and Values

Most FIT keywords and values can be abbreviated. For example, the abbreviation for $ACCESS_AND_SHARE_MODES is $AASM. Abbreviations are indicated with the word **or**. For example, the description of the $ACCESS_AND_SHARE_MODES keyword is titled $ACCESS_AND_SHARE_MODES or $AASM.

Abbreviations for FIT values are indicated in the same way. For example, one of the value descriptions for $AUTOMATIC_UNLOCK is titled 'TRUE' or 'T' or 'Yes' or 'Y' or 'ON'.

FIT Keywords and Values: Quick Reference

The following section describes all of the file information table keywords in quick reference format.

The **Input** section describes the values that you supply when you specify the associated FIT keyword on an FILEIS, FILEDA, STOREF, or OPENM call.

The **Output** section describes the values you can receive when you specify the associated FIT keyword on an IFETCH call.

The **Default** section describes the default value for the FIT keyword.

$ACCESS_AND_SHARE_MODES or $AASM

Purpose Defines the set of access and share modes for this instance of open (temporary attribute).

Input A string of paired sets of keywords. Each set specifies the share mode and access mode respectively. The list has the form

'**((access mode), (share mode))**, ... , *((access mode), (share mode))*'

The access mode keywords are:

READ
Read access.

SHORTEN
Shorten access.

APPEND
Append access.

MODIFY
Modify permission (file statistics are kept).

WRITE
Shorten, append, and modify access.

EXECUTE
Execute access.

ALL
Read, shorten, append, modify, write, and execute access. If you specify ALL, no other access mode keyword can be specified.

PERMITTED_ACCESS_MODES
The access modes granted to the requested attach depend on the conditions of the attachment described in the Remarks.

The share mode keywords are:

NONE
The instance of open does not allow sharing.

READ
Sharing is allowed for read access.

SHORTEN
Sharing is allowed for shorten access.

APPEND
Sharing is allowed for append access.

MODIFY
Sharing is allowed for modify access.

WRITE

Sharing is allowed for shorten, append, and modify access.

EXECUTE

Sharing is allowed for execute access.

ALL

Sharing is allowed for read, shorten, append and modify access. If you specify ALL, no other value can be selected.

REQUIRED_SHARE_MODES

Sharing requirements depend on the conditions of attachment described in the remarks.

DETERMINE_FROM_ACCESS_MODES

If the access modes include append, modify, or shorten, no sharing is allowed; otherwise, read and execute sharing is allowed.

Output A FORTRAN program should not fetch the $ACCESS_AND_SHARE_MODES value from the FIT. It is stored as an address which the program cannot use.

Default '((PERMITTED_ACCESS_MODES),(DETERMINE_FROM_ACCESS_MODES))'

Remarks

- To set the access and share modes for a file, you must store the values in the FIT before the file is opened because the values only take effect for the next instance of open. Also, the OPENM call that opens the file must specify the open_option parameter as 0 or omit the open_option parameter for the access and share mode values in the FIT to be used. If the OPENM call specifies a value other than 0 for the open_option parameter, the access and share mode values in the FIT are ignored.

- For PERMITTED_ACCESS_MODES, the access modes are as follows:

  - If the file is created by a FILEIS or FILEDA call, this value causes the file to be attached for all modes of access.

  - If the file is not currently attached within the job, this value causes attachment for only those modes of access currently in the file's access control entry. These access modes are qualified by the share requirements of other jobs that have the file cycle attached and by the value of the validation_ring attachment option.

  - If the file is already attached within the job but not currently open, this value causes the modes of access specified by the attachment to be used. These access modes are qualified by the value specified in the validation_ring attachment option.

  - If the file is already open within the job, this value causes the access modes specified by the attachment of the file within the job to be used. These access modes are qualified by the values specified in the open_share_modes and validation_ring attachment options.

- For REQUIRED_SHARE_MODES, the share modes are as follows:
  - If the file is created by the request, this value causes the file to be attached for exclusive access.
  - If the file exists but is not currently attached within the job, this value causes the file to be attached for the modes of sharing that consist of the union of the share requirement in the file's access control entry and the outstanding access modes specified by other jobs that have the file cycle attached. After the file is opened, your task may need to validate its toleration of the modes of sharing imposed by this value. Call IFETCH with the $GLOBAL_SHARE_MODES keyword to obtain the file's share mode values.
  - If the file is already attached within the job, this value causes the share modes specified by the attachment of the file cycle within the job to be used. After the file is opened, your task may need to validate its toleration of the modes of sharing imposed by this value. Call IFETCH with the $GLOBAL_SHARE_MODES keyword to obtain the file's share mode values.
- If you specify more than one set of access and share modes, the first set is considered the preferred value, and the remaining sets are considered alternatives. Each set is considered in order until one is found that satisfies the requirements of the request.
- This FIT value is used by the FILEIS, FILEDA, and STOREF calls. The associated file must not be open.
- When you specify this FIT value, it replaces the current value for $ACCESS_MODE or $ACCESS_AND_SHARE_MODES in the file information table. The new FIT value takes effect for the next instance of open.

Examples

The following call specifies read and modify access and read and modify sharing:

```
CALL STOREF (fitptr, '$AASM', '((READ, MODIFY),(READ, MODIFY))')
```

The following call specifies read access with no sharing if possible; otherwise it specifies read access with any mode of sharing:

```
CALL STOREF (fitptr, '$AASM', '((READ),(NONE)),((READ),(ALL))')
```

$ACCESS_MODE or $AM

Purpose

Defines the set of access modes allowed for this instance of open (temporary attribute).

For an existing file, all modes in the set must be in the usage mode set specified when you attached the file.

Input

Options are:

'READ'

Read access

'APPEND'

Append access

'SHORTEN'

Shorten access

'MODIFY'

Modify permission (file statistics are kept)

Output

Integer as follows:

1 Read access only (file statistics are not kept)
2 Modify, shorten, and append access
3 Read, modify, shorten, and append access
4 Modify access only
5 Append access only
6 Shorten access only
7 Read and modify access
8 Read and append access
9 Read and shorten access
10 Modify and shorten access
11 Modify and append access
12 Shorten and append access
13 Read, modify, and shorten access
14 Read, modify, and append access
15 Read, shorten, and append access

Default

If the access modes are not specified using $ACCESS_AND_SHARE_MODES or $ACCESS_MODE, the default is '((PERMITTED_ACCESS_MODES),(DETERMINE_FROM_ACCESS_MODES))'.

Remarks

- To set the access for a file, you must store the value in the FIT before the file is opened because the value only takes effect for the next instance of open. Also, the OPENM call that opens the file must specify the open_option parameter as 0 for the access mode value in the FIT to be used. If the OPENM call does not specify 0 as the open_option parameter, the access mode value in the FIT is ignored.
- These are the access modes to the keyed file required for each call.

Call	Access Modes
CLOSEM	None
DLTE	Append, shorten, and modify
FILEDA	None
FILEIS	None
FLUSHM	Append, shorten, or modify
GET	Read[1]
GETN	Read[1]
IFETCH	None
KEYLIST	Read
KLCOUNT	Read
KLSPACE	Read
LOCKF	Any[1]
LOCKK	Any[1]
OPENM	Any
PUT	Append[2]
PUTREP	Append and shorten[2]
REPLC	Append and shorten[2]
REWND	Any
RMKDEF	Append, shorten, and modify
RSBUILD	Read
RSGETN	Read
SKIP	Any
STARTM	Read
STOREF	None
UNLOCKF	Any
UNLOCKK	Any

[1]If an Exclusive_Access lock is requested, shorten or append access is required.

[2]If one or more alternate keys are defined in the file, append, shorten, and modify access modes are required to update the alternate indexes.

- You can specify two or more values by enclosing the values in a single pair of apostrophes and separating the values with a comma.

NOTE

No spaces can separate the values, only a comma.

For example, the following STOREF call specifies read and modify access:

```
CALL STOREF (fitptr, '$AM', 'READ,MODIFY')
```

- Specifying a string of blanks requests no access modes. But, if you request no access modes, you cannot open the file.
- Although you can store another $ACCESS_MODE value while the file is open, the new value does not take effect until the next open.
- You can also use the $ACCESS_AND_SHARE_MODES FIT value to specify access modes for an instance of open. This FIT value gives you more flexibility and also allows you to specify share modes for an instance of open.
- The access modes you specify with $ACCESS_MODE replace any previous specification of access modes with the $ACCESS_MODE or $ACCESS_AND_SHARE_MODE FIT values.

  If you specify $ACCESS_MODE, the share mode used is DETERMINE_FROM_ACCESS_MODES.

$AUTOMATIC_UNLOCK or $AU

Purpose Indicates whether a lock should be cleared automatically. This value is used when a lock is requested.

Input Options are:

'TRUE' or 'T' or 'YES' or 'Y' or 'ON'

The lock is cleared when the task issues a request for the instance of open (other than an IFETCH or STOREF call).

'FALSE' or 'F' or 'NO' or 'N' or 'OFF'

The lock is not cleared automatically. The lock is cleared by an UNLOCK call for the key value or when the instance of open closes.

Output Integer as follows:

-1 Automatic unlock is requested (YES).

0 Automatic unlock is not requested (NO).

Default 'YES'

Remarks

- This FIT value may be used by LOCKK, GET, and GETN calls as follows:
  - Used by LOCKK if the automatic_unlock parameter is omitted from the call.
  - Used when a GET or GETN call requests a lock, that is, when the FIT value $GET_AND_LOCK is YES (-1).

  NOTE

  Automatic unlock cannot be used with Preserve_Content lock intent.

- For an update request for the locked record, the automatic unlock does not occur until the operation completes. For all other requests, the automatic unlock occurs as soon as the request is issued.

$AVERAGE_RECORD_LENGTH or $ARL

Purpose

Defines the estimated median record length, in bytes, of the data records to be stored in the file. (The length should not include the primary-key length if the primary key is nonembedded.)

NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.

Input

Integer from 1 through 65497.

Output

A FORTRAN program should not fetch the $AVERAGE_RECORD_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default

None. If the FIT value is zero when a new file is opened, NOS/VE uses the arithmetic mean of the minimum and maximum record lengths as the average record length when selecting the block size for the new file.

Remarks

- When the file contains variable-length records, you should choose the average record length value as follows:
  - If almost all records in the file are nearly the same length, use that length.
  - If the record lengths are well-distributed, use the median record length.

$COLLATE_TABLE or $CT

Purpose Defines the collation table for the primary key.

The value is used only when a new file is opened for the first time; it is not preserved with the file.

Input 256-character variable. Each character in the variable is the collating weight for the corresponding ASCII character. For example, the first character in the variable is the collating weight for the first ASCII character [code 00].

Output A FORTRAN program should not fetch the $COLLATE_TABLE value from the FIT. It is stored as an address which the program cannot use.

Default None. A collation table must be specified if the primary key type is collated. The collation table can be specified by the $COLLATE_TABLE or $COLLATE_TABLE_NAME value.

Remarks OPENM copies the table from the variable to the internal entry point AAV$DCT. It then stores AAV$DCT as the collation table name and the collation table at AAV$DCT as the collation table for the new file.

$COLLATE_TABLE_NAME or $CTN

Purpose Defines the collation table for the primary key specified as the name of an entry point (preserved attribute). The value is used only when a new file is opened for the first time.

Input 1- to 31-character string specifying the entry point name of the collation table.

Output The first 8 characters of the entry point name. The name is returned left-justified, blank-filled, and in uppercase letters.

Default None. A collation table must be specified if the primary-key type is collated. The collation table can be specified by the $COLLATE_TABLE or $COLLATE_TABLE_NAME value.

Remarks

- The collation table can be one of the NOS/VE predefined collation tables. The predefined collation tables are listed in appendix D, Creating a Collation Table.
- The COLSEQ routine can be used to create a table named FTV$USER_COLLATE_TABLE which can be specified as the $COLLATE_TABLE_NAME. For information on creating a collation table, see appendix D, Creating a Collation Table.
- The entry point can be in a module already loaded with the FORTRAN program or in a module in an object library in the program-library list. For a module to be loaded from an object library, it must be in the program-library list. For more information, see the NOS/VE Object Code Management Usage manual.

$COMPRESSION_PROCEDURE_NAME or $CPN

Purpose Defines the data compression or encryption procedure (preserved attribute).

Input 1- to 31-character string specifying an entry point name in an object library in the current program library list.

The name must be enclosed in apostrophes ('name').

Output First 8 characters of the entry point name. The name is returned left-justified, blank-filled, and in uppercase letters.

Default None. Unless a procedure is specified when the file is created, no compression procedure is used.

Remarks

- This FIT value can be specified only before the file is opened for the first time. The value is stored as a preserved attribute when the file is first opened.

- The compression procedure is not stored with the file. It must be loaded each time the file is opened. Therefore, the object library containing the compression procedure must be in the program library list.

  For example, this command adds a library to the library list:

  ```
  /set_program_attributes, add_library=$user.my_obj_library
  ```

- A compression procedure name AMP$RECORD_COMPRESSION is provided with the system. It compresses strings of consecutive ASCII spaces, ASCII zeros, binary zeros, and nulls.

  When records are fetched from the file, AMP$RECORD_COMPRESSION decompresses the record data to its original length and content.

  (The system-defined procedure is on a system library so you do not need to add it to your program library list.)

  For Better Performance

  Usually a compression procedure individually processes each byte of data for each record operation. This significantly increases the time required for each record operation. Therefore, you should specify a compression procedure only when the special processing it performs is worth the extra processing time.

- If you specify a compression procedure, you should consider its effect when specifying the file structure attributes. If you specify an $AVERAGE_RECORD_LENGTH value, it should be the average record length after data compression. Similarly, when creating a direct-access file, you should choose the INITIAL_HOME_BLOCK_COUNT value based on the size of the compressed file data.

- User-defined compression procedures can be written, but the procedures must be written in the CYBIL language. For more information, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.
- The NOS/VE compression procedure performs both compression and decompression.
- The primary-key field can be anywhere in the record.

$DATA_PADDING or $DP

Purpose Percentage of block space left empty as each data block is created during the first instance of open of an indexed-sequential file (preserved attribute).

Input Integer from 0 through 99. The padding percentage must allow at least one maximum-length record to be written to each block.

Output A FORTRAN program should not fetch the $DATA_PADDING value from the FIT. It is stored as an address which the program cannot use.

Default 0 (no data block padding).

Remarks Data block padding leaves space for insertion of additional data in the future without the need for additional index levels. Fewer index levels mean less access time to find a record. However, data block padding could result in unused file space.

$DELETE_DATA or $DD

Purpose Specifies whether or not the data in the file is deleted as a result of the instance of open.

Input Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

Deletes contents of the file.

'FALSE', 'F', 'NO', 'N', 'OFF'

Saves contents of the file.

Output A FORTRAN program should not fetch the $DELETE_DATA value from the FIT. It is stored as an address which the program cannot use.

Default 'FALSE'

Remarks

- Data is deleted from your file cycle at the time it is opened if a value of TRUE (or alias) is specified and if the following conditions are met:

  1. The file must be currently attached to the job for exclusive access (no sharing) either by a previous attachment or by the current request.
  2. The access modes specified for the instance of open must include shorten.
  3. The file must be currently unopened within the job.
  4. The file must be opened at its beginning-of-information position.

- After the data in a file is deleted, the file's access control entry and file's mass storage space are not released. Also, the file cycle remains registered in the appropriate catalog.

DX (Data Exit Procedure)

Purpose End-of-data exit procedure.

Input Name of a subroutine that is declared as EXTERNAL.

Output A FORTRAN program should not fetch the DX value from the FIT. It is stored as an address which the program cannot use.

Default None.

Remarks

- If a DX value has been stored in the FIT, a GETN or SKIP call calls the specified subroutine when the GETN or SKIP call encounters the beginning-of-information or end-of-information.
- The data-exit routine can determine whether the file is at its BOI or EOI by fetching the $FILE_POSITION value.

$EMBEDDED_KEY or $EK

Purpose Indicates whether the primary key is embedded or nonembedded (preserved attribute).

Input Options are:

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'

Embedded key. (The key value is part of the record data.)

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'

Nonembedded key. (The key value is separate from the record data.)

Output Integer as follows:

-1 Embedded key. (The key value is part of the record data.)

0 Nonembedded key. (The key value is separate from the record data.)

Default 'YES'

$ERROR_COUNT or $EC

Purpose Number of nonfatal errors that have been returned by keyed-file interface calls since the OPENM call.

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer. The value is limited by a nonzero $ERROR_LIMIT value.

Default Initialized to 0 when the file is opened.

Remarks This attribute can be fetched only while the file is open.

$ERROR_EXIT_PROCEDURE_NAME or $ERROR_EXIT_NAME or $EEPN or $EEN

Purpose Error-exit procedure (temporary attribute).

Input 1- to 31-character string specifying the entry point name of the error_exit procedure name. The name must be enclosed in apostrophes ('name').

Output The first 8 characters of the entry point name. The name is left-justified, blank-filled, and in uppercase letters.

Default None. If you do not specify a name before opening the file, the system does not load an error-exit procedure.

Remarks

- The error-exit entry point may be an entry point already loaded with the program or an entry point in an object library. For a module to be loaded from an object library, it must be in the program library list. For more information on program libraries and the SET_PROGRAM_ATTRIBUTES command, see the NOS/VE Object Code Management Usage manual.

- You can clear the $ERROR_EXIT_PROCEDURE_NAME value by calling STOREF with a string of blanks. For example:

```
CALL STOREF (FITPTR, '$ERROR_EXIT_PROCEDURE_NAME', '  ')
```

- The OPENM call gets the address of the $ERROR_EXIT_PROCEDURE_NAME procedure and stores it as the $ERROR_EXIT_PROCEDURE value in the FIT. Therefore, if you specify two error-exit procedures before opening the file: one using the $ERROR_EXIT_PROCEDURE_NAME keyword and the other, the $ERROR_EXIT_PROCEDURE keyword, the procedure specified using $ERROR_EXIT_PROCEDURE_NAME is used.

- Storing an $ERROR_EXIT_PROCEDURE_NAME value after the file is open has no effect; the value is used only if the file is re-opened using the same FIT.

  To change the error-exit procedure for the current open, specify an error-exit procedure parameter on a keyed-file interface call or specify the $ERROR_EXIT_PROCEDURE value on a STOREF call.

$ERROR_EXIT_PROCEDURE or $EEP

Purpose Error-exit procedure.

Input Name of a subroutine that is declared as EXTERNAL in the calling program.

Output A FORTRAN program should not fetch the $ERROR_EXIT_PROCEDURE value from the FIT. It is stored as an address which the program cannot use.

Default None. The error-exit procedure specified by the $ERROR_EXIT_PROCEDURE_NAME attribute before the file was opened (if any) is used.

Remarks

- Specifying a value using the $ERROR_EXIT_PROCEDURE keyword changes the effective error-exit procedure immediately. (Specifying a value using the $ERROR_EXIT_PROCEDURE_NAME keyword changes the procedure only when the file is opened.)
- A nonzero value specified with the $ERROR_EXIT_PROCEDURE keyword is stored as the default error-exit procedure value in the FIT. It becomes the default eep parameter value until another eep value is specified on a call.
- To clear the $ERROR_EXIT_PROCEDURE value, call STOREF to store a value of zero as the $ERROR_EXIT_PROCEDURE value.

$ERROR_LIMIT or $EL

Purpose Nonfatal error limit (temporary attribute). When the limit is reached, a fatal error is returned.

Input Integer between 0 and 65535. 0 allows unlimited trivial errors.

Output A FORTRAN program should not fetch the $ERROR_LIMIT value from the FIT. It is stored as an address which the program cannot use.

Default 0 (no limit).

Remarks ERROR_LIMIT is compared to ERROR_COUNT to determine when the error limit has been reached. For more information on error processing, see chapter 1, Keyed-File Interface Concepts.

$ERROR_STATUS or $ES

Purpose Condition code returned by the previous keyed-file interface call.

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer condition code. A zero value indicates the previous keyed-file interface call completed successfully, without error.

The condition code for an abnormal status consists of the two-byte product identifier (such as AA or AM) and a three-byte value specifying the particular error condition for that product.

To translate the condition code to a string, use the CONDSYM subprogram. Specify the condition code as the first parameter and CONDSYM returns the translated string as the second parameter and the length of the string as the third parameter.

For example:

```
C   Call FILEIS and induce an error by not
C   specifying the $MAXIMUM_RECORD_LENGTH
C   FIT keyword, which is required.
C
      CALL FILEIS (FIT_PTR,
     +'$LOCAL_FILE_NAME',    'NEW_IS_FILE',
     +'$KEY_LENGTH',          5)
C
C   Call OPENM to open the file.
C
      CALL OPENM(FIT_PTR, 'NEW', 'R')
C
C   Call IFETCH to retrieve the condition
C   code.
C
      CALL IFETCH(FIT_PTR,
     +'$ERROR_STATUS',     CONDITION_CODE)
C
C    Call CONDSYM to translate CONDITION_CODE
C    to a string CONDITION_NAME and
C    print it.
C
      CALL CONDSYM(CONDITION_CODE,
     +CONDITION_NAME, LENGTH)
      PRINT *, CONDITION_NAME
C
C    The condition name AA 3210 is printed.
C
```

The CONDSYM subprogram is described under NOS/VE Status Subprograms in the FORTRAN Version 1 or Version 2 Language Definition manuals.

Default Initialized to 0 before each keyed-file interface call.

Remarks

- If an error-exit procedure has not been specified, the program should fetch the error status value after each keyed-file interface call. A nonzero value returned indicates that the call did not complete successfully.
- The NOS/VE Diagnostic Messages manual lists the meaning of each condition name. To access the manual online, enter:

```
/help manual=messages
```

Then use the INDEX function on the condition name.

$ESTIMATED_RECORD_COUNT or $ERC

Purpose Estimated number of data records to be stored in the file.

NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.

Input Integer from 1 through 4398046511103 (2**42 - 1).

Output A FORTRAN program should not fetch the $ESTIMATED_RECORD_COUNT value from the FIT. It is stored as an address which the program cannot use.

Default The $RECORD_LIMIT value, if specified. If no $RECORD_LIMIT value is specified, an estimate of 100,000 records is used.

$FILE_IDENTIFIER or $FI

Purpose Returns the CYBIL file identifier for the current open of the file.

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer.

Remarks

- An IFETCH call can fetch the file identifier only while the file is open. The file identifier cannot be fetched before the OPENM call or after the CLOSEM call.
- A FORTRAN program fetches the file identifier so that it can pass it to a CYBIL procedure. The CYBIL procedure requires the file identifier so that it can issue file interface calls for the open file.
- To receive the file identifier value as a parameter, a CYBIL procedure declaration specifies a VAR declaration of type AMT$FILE_IDENTIFIER. For example:

```
PROCEDURE cybil_proc (VAR fi: amt$file_identifier);
```

- The CYBIL procedure must not close a file opened in the FORTRAN program. A file opened by an OPENM call must be closed by a CLOSEM call (or by program termination). Otherwise, the results of the file operations are undefined.
- File interface calls made outside the FORTRAN program do not update the FIT. The CYBIL subprogram should not call AMP$STORE to change file attribute values because the changed values are not copied to the FIT. Subsequent calls to IFETCH would then return out-of-date information.

$FILE_ORGANIZATION or $FO

Purpose File organization (preserved attribute). The file organization determines the method of storing and accessing file data.

Input Options are:

'INDEXED_SEQUENTIAL' or 'IS'
Indexed-sequential file organization

'DIRECT_ACCESS' or 'DA'
Direct access file organization

Output Integer as follows:

3 Indexed-sequential file organization

5 Direct access file organization

Default Set by the call that created the FIT. FILEIS sets the file organization to indexed-sequential; FILEDA sets the file organization to direct access.

$FILE_POSITION or $FP

Purpose Indicates the position of the file after the last keyed-file interface call (returned attribute).

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer as follows:

1 File is positioned at the beginning-of-information (BOI).

8 File is positioned at the end of a key list (returned only if an alternate key is currently selected).

16 File is positioned at the end of a record (EOR), but not at the end of a key list (returned only if an alternate key is currently selected.)

64 File is positioned at the end-of-information (EOI).

Default When the file is opened, but before any records are processed, $FILE_POSITION has the same value as $OPEN_POSITION. The default $OPEN_POSITION value is $BOI.

$FORCED_WRITE or $FW

Purpose Indicates when modified blocks of the file are to be written to mass storage (preserved attribute).

Input Options are:

'TRUE' or 'T' or 'YES' or 'Y' or 'ON'

The system writes each modified block to mass storage immediately after the modification.

'FORCED_IF_STRUCTURE_CHANGE' or 'FISC'

The system writes modified blocks to mass storage immediately if the change affects more than one block.

'FALSE' or 'F' or 'NO' or 'N' or 'OFF'

The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy.

Output Integer as follows:

-1 The system writes each modified block to mass storage immediately after the modification (TRUE).

0 The system writes modified blocks to mass storage immediately if the change affects more than one block (FORCED_IF_STRUCTURE_CHANGE).

+1 The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy (FALSE).

Default 'FALSE'. (The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy.)

Remarks

- You can request that the entire file be copied to disk by calling FLUSHM. FLUSHM copies internal tables as well as data and index blocks. (A $FORCED_WRITE copy does not copy internal tables.)
- If the file could be shared and the $FORCED_WRITE value is either -1 or 0, the block size of the file should be a multiple of the system page size.

  This ensures that multiple opens are not updating blocks in the same page. Otherwise, a forced-write operation could write a page that contains partially-altered blocks. (A warning message is issued if this possibility exists.)

FNF (Fatal/Nonfatal Flag)

Purpose Indicates the severity of the last error for the file as fatal or nonfatal.

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer values as follows:

0 The error severity is nonfatal.

−1 The error severity is fatal.

Remarks This value is not defined outside the FORTRAN keyed-file interface. No NOS/VE keyword is defined for the FIT value.

$GET_AND_LOCK or $GAL

Purpose Indicates whether a GET or GETN call issues a lock request for the key value before reading the record.

Input Options are:

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'

A GET or GETN call requests a lock.

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'

A GET or GETN call does not request a lock.

Output Integer as follows:

-1 A GET or GETN call requests a lock (YES).

0 A GET or GETN call does not request a lock (NO).

Default 'NO' (a GET or GETN call does not request a lock).

Remarks These FIT values are used as parameter values for the lock if the $GET_AND_LOCK value is YES (-1):

$AUTOMATIC_UNLOCK
$LOCK_INTENT
$WAIT_FOR_LOCK

For more information on locks, see chapter 3, Sharing Keyed Files.

$GLOBAL_ACCESS_MODES or $GAM

Purpose Returns the set of access modes in effect for the instance of attach of an existing file (returned attribute).

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer as follows:

0 No access
1 Read access only (file statistics are not kept)
2 Modify, shorten, and append access
3 Read, modify, shorten, and append access
4 Modify access only
5 Append access only
6 Shorten access only
7 Read and modify access
8 Read and append access
9 Read and shorten access
10 Modify and shorten access
11 Modify and append access
12 Shorten and append access
13 Read, modify, and shorten access
14 Read, modify, and append access
15 Read, shorten, and append access
16 Execute access only
17 Read and execute access (file statistics are not kept)
18 Modify, shorten, append, and execute access
19 Read, modify, shorten, append, and execute access
20 Modify and execute access
21 Append and execute access
22 Shorten and execute access
23 Read, modify, and execute access
24 Read, append, and execute access
25 Read, shorten, and execute access
26 Modify, shorten, and execute access
27 Modify, append, and execute access
28 Shorten, append, and execute access
29 Read, modify, shorten, and execute access
30 Read, modify, append, and execute access
31 Read, shorten, append, and execute access

Remarks

- If the file is open, $GLOBAL_ACCESS_MODES returns the access modes that were specified when the file was opened. If you specified more than one set of access modes, $GLOBAL_ACCESS_MODES returns the access modes that actually took effect.

- If the file is an unopened permanent file, $GLOBAL_ACCESS_MODES returns the access modes for which the file may be opened.

- If the file is an unopened temporary file, $GLOBAL_ACCESS_MODES returns 19, all access modes.

$GLOBAL_SHARE_MODES or $GSM

Purpose Returns the share mode set in effect for the current instance of attach on an existing file (returned attribute).

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer as follows:

0 No sharing.
1 Sharing for read access only (file statistics are not kept)
2 Sharing for modify, shorten, and append access
3 Sharing for read, modify, shorten, and append access
4 Sharing for modify access only
5 Sharing for append access only
6 Sharing for shorten access only
7 Sharing for read and modify access
8 Sharing for read and append access
9 Sharing for read and shorten access
10 Sharing for modify and shorten access
11 Sharing for modify and append access
12 Sharing for shorten and append access
13 Sharing for read, modify, and shorten access
14 Sharing for read, modify, and append access
15 Sharing for read, shorten, and append access
16 Sharing for execute access only
17 Sharing for read and execute access (file statistics are not kept)
18 Sharing for modify, shorten, append, and execute access
19 Sharing for read, modify, shorten, append, and execute access
20 Sharing for modify and execute access
21 Sharing for append and execute access
22 Sharing for shorten and execute access
23 Sharing for read, modify, and execute access
24 Sharing for read, append, and execute access
25 Sharing for read, shorten, and execute access
26 Sharing for modify, shorten, and execute access
27 Sharing for modify, append, and execute access
28 Sharing for shorten, append, and execute access
29 Sharing for read, modify, shorten, and execute access
30 Sharing for read, modify, append, and execute access
31 Sharing for read, shorten, append, and execute access

Remarks

- If the file is open, $GLOBAL_SHARE_MODES returns the share modes that were specified when the file was opened. If you specified more than one set of share modes, $GLOBAL_SHARE_MODES returns the share modes that actually took effect.
- If the file is an unopened permanent file, $GLOBAL_SHARE_MODES returns the share modes for which the file may be opened.
- If the file is an unopened temporary file, $GLOBAL_SHARE_MODES returns 0, no share modes.

$HASHING_PROCEDURE_NAME or $HPN

Purpose Name of the hashing procedure used to hash primary-key values for the direct access file (preserved attribute).

Input 1- to 31-character string specifying an entry point in an object library in the current program library list. The name must be enclosed in apostrophes ('name').

Output First 8 characters of the entry point name. The name is left-justified, blank-filled, and in uppercase letters.

Default AMP$SYSTEM_HASHING_PROCEDURE (the system default hashing procedure).

Remarks

- This FIT value can be specified only before the file is opened for the first time. The value is stored as a preserved attribute when the file is first opened.
- A user-defined hashing procedure must be written in the CYBIL language only. For more information, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.
- The hashing procedure is not stored with the file. It must be loaded each time the file is opened. Thus, the object library containing the hashing procedure must be in the program library list.

  For Better Performance

  Although any ring-attributes value is valid for the object library containing the hashing procedure, you should store the hashing procedure in a ring 4 object library.

  This improves performance because hashing procedures are executed as asynchronous tasks. (Usually, site personnel maintain the ring 4 object libraries.)

- A hashing procedure can be specified by name only; it cannot be specified by address.

$INDEX_LEVELS or $INDEX_LEVEL or $IL

Purpose For a new indexed-sequential file, the target number of index levels or, for an existing indexed-sequential file, the current number of index levels.

Input Target number of index levels (integer from 0 through 15). The system uses this value as a guideline in its selection of the block size for a new file.

Output Current number of index levels (integer from 0 through 15). An empty file has 0 index levels.

Default For a new file, 2 index levels.

Remarks

- If specified before the file is created, NOS/VE uses the INDEX_LEVELS value when selecting the block size for a new file if the maximum block length for the file is not specified. The specified value is not preserved with the file because it is used only when the file is opened for the first time.
- For an existing file, the value returned is the current number of levels of indexing in the indexed-sequential file.
- The current number of index levels can be fetched only while the file is open.

$INDEX_PADDING or $IP

Purpose Percentage of block space left empty in each index block created during the first instance of open of the file (preserved attribute).

Input Integer from 0 to 99. The padding percentage must allow at least three index records to be written to the block. (The index record length is the primary key length plus 4 bytes.)

Output A FORTRAN program should not fetch the $INDEX_PADDING value from the FIT. It is stored as an address which the program cannot use.

Default 0 (no index block padding).

$INITIAL_HOME_BLOCK_COUNT or $IHBC

Purpose Number of home blocks in the direct access file (preserved attribute).

Input Integer from 1 through 4387945511193 (2**42 – 1).

Output A FORTRAN program should not fetch the $INITIAL_HOME_BLOCK_COUNT value from the FIT. It is stored as an address which the program cannot use.

Default None. You must specify a value for this attribute when defining a new direct-access file.

Remarks

- This value specifies the number of blocks allocated for the new direct access file. The blocks should accommodate all records expected to be written to the file. The addition of more records would require allocation of overflow blocks, slowing access to the overflow records.
- The initial_home_block_count should allow for a loading factor of no more than 90%. In other words, allocate at least 10% extra space in the file because the hashing procedure may not uniformly distribute records among the home blocks.
- For best results, the initial_home_block_count should be a prime number.
- For more information, see the discussion of direct-access files in chapter 1, Keyed-File Interface Concepts.

$KEY_ADDRESS or $KA

Purpose Location of the key value.

Input Variable name.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the $KEY_ADDRESS value from the FIT. It is stored as an address which the program cannot use.

Remarks

- A key address is required in these cases:
  - When a PUT call writes a record with a nonembedded key.
  - When a GET call reads a record by its primary-key value.
  - For any STARTM or LOCKK call.
- A key address is optional for a GET call when an alternate key is selected. GET reads the alternate-key value from the key address if a key_area value is specified on the call or in the FIT.
- The $KEY_ADDRESS value in the FIT is used when 0 is specified as the $KEY_ADDRESS parameter on a call.
- If a keyed-file interface call specifies a $KEY_ADDRESS value, the value is copied to the FIT. It becomes the default value for subsequent calls.

$KEY_LENGTH or $KL

Purpose Key length (preserved attribute). It is the primary-key length for a new file. For an existing file, it is the key length when selecting a key by position and length.

Input For an embedded key (primary or alternate), an integer from 1 through 255, but not greater than the minimum record length.

For a nonembedded primary key, an integer from 1 through 255.

For an integer key, an integer from 1 through 8.

Output A FORTRAN program should not fetch the $KEY_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default None. You must specify the primary-key length before calling OPENM for a new file.

$KEY_NAME or $KN

Purpose Name of the selected key.

Input 1- to 31-character string specifying the key name. The name of the primary key is $PRIMARY_KEY.

Output The first 8 characters of the key name. The name is returned left-justified, blank-filled, and in uppercase letters.

Default The primary key ($PRIMARY_KEY).

Remarks

- A key name can be specified by the OPENM call or by a STOREF call while the file is open. It cannot be specified by the FILEIS or FILEDA call or by a STOREF call before the OPENM call or after the CLOSEM call.
- The name of an alternate key is defined when the key is defined. For more information, see chapter 2, Alternate Keys.

$KEY_POSITION or $KP

Purpose Byte position at which the key begins (preserved attribute).

It is the position of the primary key for a new file. For an existing file, it is the key position used when selecting a key by position and length.

Input Integer from zero to the maximum record length for the file. However, the key position value added to the key length value must not exceed the minimum record length.

The byte positions in a record are numbered from the left, beginning with zero.

Output A FORTRAN program should not fetch the $KEY_POSITION value from the FIT. It is stored as an address which the program cannot use.

Default Zero. If the key is embedded, the key is assumed to begin at the leftmost byte of the record. If the key is nonembedded, the key position value is not used.

$KEY_RELATION or $KR

Purpose Relation between the key value in the record and the key value at the $KEY_ADDRESS location.

Input Options are:

'EQUAL_KEY' or 'EK'

The record key value must be equal to the specified key value.

'GREATER_OR_EQUAL_KEY' or 'GOEK'

The record key value must be greater than or equal to the specified key value.

'GREATER_KEY' or 'GK'

The record key value must be greater than the specified key value.

Output Integer as follows:

1 The record key value must be equal to the specified key value.

3 The record key value must be greater than or equal to the specified key value.

6 The record key value must be greater than the specified key value.

Default EQUAL_KEY. (The key value in the record must be equal to the specified key value.)

Remarks

- The $KEY_RELATION value is used only by GET and STARTM calls. A GET call reads the first record that satisfies the relation. A STARTM call positions the file at the first record that satisfies the relation.
- The $KEY_RELATION FIT value is not used by calls to a direct-access file while its primary key is selected (because no index with ordered key values exists for the key).

$KEY_TYPE or $KT

Purpose Primary key type for a new indexed-sequential file (preserved attribute).

Input Options are:

'COLLATED' or 'C'

A key value is a string of characters; it is sorted byte-by-byte according to a user-specified collating sequence.

'INTEGER' or 'I'

A key value is a signed integer (8 bytes long); it is sorted in ascending numerical order.

'UNCOLLATED' or 'U'

A key value is a string of characters; it is sorted byte-by-byte according to the default ASCII collating sequence.

Output Integer as follows:

1 A key value is a string of characters; it is sorted byte-by-byte according to a user-specified collating sequence.

2 A key value is a signed integer (8 bytes long); it is sorted in ascending numerical order.

3 A key value is a string of characters; it is sorted byte-by-byte according to the default ASCII collating sequence.

Default Uncollated keys ('U').

Remarks The primary-key type for a direct access file is always uncollated, regardless of the specified value. (The primary-key values are not sorted so a sort-order specification is irrelevant.)

$LAST_OPERATION or $LO

Purpose Most recent keyed-file interface call for the file (returned attribute).

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output One of the following integers:

0 FILEIS (FIT created for an indexed-sequential file)
1 OPENM (open request)
2 CLOSEM (close request)
3 GET (random read request)
4 GETN (sequential read request)
5 PUT (write request)
8 DLTE (delete request)
9 REPLC (replace request)
10 REWND (rewind request)
11 PUTREP (put/replace request)
12 SKIP (skip forward request)
13 SKIP (skip backward request)
14 STARTM (start request)
19 RMKDEF (alternate-key definition request)
20 KLCOUNT (key-list count request)
21 KLSPACE (key-list block count request)
22 KEYLIST (key list request)
23 LOCKF (lock file request)
24 LOCKK (key value lock request)
25 UNLOCKF (clear file lock request)
26 UNLOCKK (clear key value lock request)
27 FILEDA (FIT created for a direct-access file)
28 FILESK (not implemented yet)
29 RSBUILD (result set build request)
30 RSGETN (result set get next request)

Remarks The following calls do not change the $LAST_OPERATION value in the FIT. After one of these calls, IFETCH returns the value of the preceding keyed-file interface call.

- IFETCH, FLUSHM, and STOREF
- The parcel calls (PBEGIN, PABORT, PCOMMIT, PDETERM, and PPUTMSG)
- The result set calls (other than RSBUILD and RSGETN)

$LOCAL_FILE_NAME or $LFN

Purpose Name of the file in the $LOCAL catalog.

Input File name. For a new file, the file cannot already exist. For an existing file, the file must exist. (It can be a temporary file or an attached permanent file.)

Output The first 8 characters of the file name. The name is returned left-justified, blank-filled, and in uppercase letters.

Default None. This is a required parameter; it must be specified by the FILEIS or FILEDA call that creates the FIT or a STOREF call before the file is opened.

Remarks

- If the old/new flag (ON) is set to 'OLD', OPENM searches for a file with the specified name in the working catalog. If the old/new flag (ON) is set to 'NEW', OPENM attempts to create a file with the specified name in the working catalog.
- A FORTRAN program must set the $LOCAL_FILE_NAME (LFN) value in the FIT before calling OPENM. If the $LOCAL_FILE NAME value has not been specified, the OPENM call returns a fatal error.
- The LOCAL_FILE_NAME value cannot be changed while the file is open.

$LOCK_EXPIRATION_TIME or $LET

Purpose Number of milliseconds between the time a lock is granted and the time that it could expire (preserved attribute).

Input Integer from 0 through 604,800,000. (0 specifies an unlimited expiration time.)

Output A FORTRAN program should not fetch the $LOCK_EXPIRATION_TIME value from the FIT. It is stored as an address which the program cannot use.

Default 60,000 milliseconds (60 seconds).

Remarks

- An expired lock prevents further access to the file by the owner of the lock. To remove an expired lock, the owner must call UNLOCKK or close the instance of open.
- Although the lock expiration time is an attribute preserved with the file after its first open, the attribute value can be changed by the NOS/VE command, CHANGE_FILE_ATTRIBUTE.
- To read about lock expiration, see Lock Expiration and Clearing in chapter 3, Sharing Keyed Files.

$LOCK_INTENT or $LI

Purpose Purpose of the lock.

Input Options are:

'PRESERVE_CONTENT' or 'PC'
Preserve_Content for reading.

'PRESERVE_ACCESS_AND_CONTENT' or 'PAAC' or 'PAC'
Preserve_Access_and_Content for reading and possibly updating.

'EXCLUSIVE_ACCESS' or 'EA'
Exclusive_Access for updating (requires shorten or append access).

Output Integer as follows:

0 PRESERVE_CONTENT

1 PRESERVE_ACCESS_AND_CONTENT

2 EXCLUSIVE_ACCESS

Default PRESERVE_ACCESS_AND_CONTENT.

Remarks

- This FIT value is used when:
  - A GET or GETN call requests a lock, that is, when the FIT value $GET_AND_LOCK is YES (-1).
  - A LOCKF and LOCKK if the li parameter is omitted from the call.
- A PRESERVE_CONTENT lock cannot be automatically unlocked. Also, a PRESERVE_CONTENT lock must be cleared before the lock_intent for the lock can be changed to PAC or EA.
- An EXCLUSIVE_ACCESS lock is allowed only if the instance of open has shorten and/or append access to the file.

$LOG_RESIDENCE or $LR

Purpose Catalog in which the update recovery log for the keyed file is written (preserved attribute).

Input Variable containing the path to the log catalog.

Output First 8 characters of the log catalog path. The path is returned left-justified, blank-filled, and in uppercase letters.

Default None if the $LOGGING_OPTIONS value does not include M (enabling media recovery); otherwise, the default is $SYSTEM.AAM.SHARED_RECOVERY_LOG.

Remarks

- The specified log must have been previously created using the Administer_Recovery_Log utility. (The default log is created during system installation.)
- Whenever you change the log residence attribute of an existing file to a log other than the default log, you should immediately backup the keyed file. Otherwise, if the file is damaged, the RECOVER_FILE_MEDIA option of the Recover_Keyed_File utility cannot execute successfully for the file.
- The Administer_Recovery_Log utility and Recover_Keyed_File utility descriptions are in the NOS/VE Advanced File Management Usage manual.

$LOGGING_OPTIONS

Purpose Options enabling use of parcels and keyed-file recovery options (preserved attribute).

Input Options are:

'ENABLE_PARCELS' or 'EP'

Allows use of parcels when updating keyed files (see description of Parcels in chapter 4).

'ENABLE_MEDIA_RECOVERY' or 'EMR'

The system maintains an update recovery log for the keyed file. (Update recovery logs are described in the NOS/VE Advanced File Management Usage manual.)

'ENABLE_REQUEST_RECOVERY' or 'ERR'

When a task aborts, the automatic close removes any partially-completed update operation.

'ALL'

All logging options are specified.

Output A FORTRAN program should not fetch the $LOGGING_OPTIONS value from the FIT. It is stored as an address which the program cannot use.

Default No logging options selected.

Remarks

- Multiple options can be specified in any order.
- While the logging options include ENABLE_PARCELS, an attempt to open the keyed file allowing write concurrency must include modify access in its access mode set. If no sharing is allowed or only read sharing, modify access is not required.

$MAJOR_KEY_LENGTH or $MKL

Purpose

Length of the key value to be used by the next STARTM or GET call. The location of the key value is given by the $KEY_ADDRESS value.

For a fixed-length key, the value is the major-key length, the number of leftmost key-value bytes compared.

For a variable-length key, the value is the length of the key specified by the call.

Input

Integer from 0 through the key length value.

Output

A FORTRAN program should not fetch the $MAJOR_KEY_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default

For a fixed-length key, 0 (the full key value is used).

For a variable-length key, the key length is required so a nonzero value must be specified.

Remarks

- The $MAJOR_KEY_LENGTH FIT value is not used by calls to a direct-access file while its primary key is selected (because no index with ordered key values exists for the key).
- When using a major key for a fixed-length key, the call compares only the leftmost bytes of the key value.
- For a variable-length alternate key, the key value is compared with the full alternate-key value stored in the index, not just the leftmost bytes.
- The $MAJOR_KEY_LENGTH value is reset to zero after execution of the STARTM or GET call that uses the value.
- Major-key use with an integer key is not recommended. The leftmost bytes of an integer value are seldom meaningful beyond indicating the sign of the value.

$MAXIMUM_BLOCK_LENGTH or $MAXBL

Purpose Block length, in bytes, for a new file (preserved attribute).

Input Integer from 1 through 65,536. If the value is less than the maximum record length, it is increased to that value. Then, it is increased, if necessary, to the next power of 2 from 2,048 through 65,536.

If the specified value is less than the maximum record length, it is increased to that value. Then, if the value is not a power of 2 between 2048 and 65536, it is changed as follows:

- A value less than 2048 is increased to 2048 (the minimum allocation unit).
- A value between 2048 and 65536, but not a power of 2, is increased to the next power of 2 (4096, 8192, 16384, 32768, or 65536).
- A value greater than 65536 is decreased to 65536.

Output A FORTRAN program should not fetch the $MAXIMUM_BLOCK_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default The system selects the block length using the AVERAGE_RECORD_LENGTH, ESTIMATED_RECORD_COUNT, INDEX_LEVELS, and RECORDS_PER_BLOCK values, if specified. The minimum block length selected by the system is 1 page.

Remarks

- If the file could be changed by more than one instance of open at the same time and forced-writing will be used (the $FORCED_WRITE attribute is -1 [TRUE] or 0 [FORCED_IF_STRUCTURE_CHANGE]), the block size should be a multiple of the system page size.
- This ensures that more than one instance of open is not updating blocks in the same page; otherwise, a forced-write operation could write a page to mass storage that contains partially-altered blocks. (A warning message is issued if this situation exists.)

$MAXIMUM_RECORD_LENGTH or $MAXRL

Purpose Maximum record length, in bytes, for a new file (preserved attribute).

Input Integer from 1 through 65497.

Output A FORTRAN program should not fetch the $MAXIMUM_RECORD_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default None. You must specify the maximum record length when creating a new keyed file.

$MESSAGE_CONTROL or $MC

Purpose Indicates the additional information written to the $ERRORS file (temporary attribute).

Input Options are:

'MESSAGES' or 'M'
Informative messages.

'STATISTICS' or 'S'
Statistic messages.

'TRIVIAL_ERRORS' or 'T'
Trivial (nonfatal) error messages.

' ' (one or more blanks)
No additional information (fatal and catastrophic messages only).

Output Integer as follows:

Default 0 (no additional information). Only fatal and catastrophic error messages are written to the $ERRORS file.

Remarks

- It is recommended that you request at least informative and trivial (nonfatal) error messages.
- To specify two or more values, enclose the values in a single pair of apostrophes; a comma is required between values. (Spaces are also allowed between values.)

$MINIMUM_RECORD_LENGTH or $MINRL

Purpose Minimum record length, in bytes, for a new file (preserved attribute).

Input Integer from 0 through 65497 bytes. The value must be less than or equal to the maximum record length.

Output A FORTRAN program should not fetch the $MINIMUM_RECORD_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default For fixed-length records, the default value is 0; however, the length of each fixed-length record must be the $MAXIMUM_RECORD_LENGTH value.

For variable-length records with an embedded primary key, the default value is the sum of the key position and key length values. For variable-length records with a nonembedded primary key, the default value is 1 byte.

Remarks For variable-length records, it is recommended that you explicitly specify the minimum record length. The minimum record length must include the primary-key field and any alternate-key fields (or corresponding sparse-key control characters).

$NESTED_FILE_NAME or $NFN

Purpose Name of the selected nested file.

Input Name of an existing nested file in the file. (The FORTRAN keyed-file interface cannot create a new nested file.)

Output First 8 characters of the nested-file name. The name is returned left-justified, blank-filled, and in uppercase letters.

Default $MAIN_FILE (the default nested file).

Remarks

- Storing a nested-file name in the FIT selects that nested file for use. All following calls operate on the selected nested file until another nested-file name is stored.

- A nested-file name can be specified only while the file is open. It cannot be specified by the FILEIS or FILEDA call or by a STOREF call before the OPENM call or after the CLOSEM call.

- When a nested file is selected for the first time during an instance-of-open, its open position is specified by the $OPEN_POSITION attribute of the file.

  Later re-selection of the nested file during the instance-of-open positions the file at its last position during its prior selection. Thus, a task can sequentially access records in one nested file, select another nested file, re-select the first nested file, and continue the sequential access.

- The first time a nested file is selected during an instance-of-open, the first key selected is the primary key.

  Later, when a nested file is re-selected during the instance-of-open, the selected key is set to the last key selected during the previous selection of the nested file. Thus, a task can select an alternate key, select another nested file, re-select the first nested file and continue use of the previously selected alternate key.

- Selection of another nested file does not release any locks.

  An expired lock status is not returned when locks expire for nested files other than the nested file currently selected. However, an expired lock status is returned if the task re-selects the nested file and attempts an operation on that nested file.

- The FORTRAN keyed-file interface cannot create additional nested files in a keyed file. To do so, use the CREATE_KEYED_FILE utility, the NOS/VE command COPY_KEYED_FILE, or a CYBIL program.

OC (Open/Close Flag)

Purpose Indicates whether a file is open or closed (returned attribute).

Input You can only fetch information with this FIT keyword. You cannot store information with it.

Output Integer as follows:

0 The file has never been opened.

1 The file is open.

2 The file is closed.

ON (Old/New Flag)

Purpose Indicates whether the next OPENM call is to create a new file or open an existing file.

Input Options are:

'OLD'

The file exists.

'NEW'

The file is being created.

Output Integer as follows:

0 The file exists.

−1 The file is being created.

Default Set to 'NEW' if the $ACCESS_MODES value is 'NEW'. Reset to 'OLD' by a CLOSEM call.

$OPEN_POSITION or $OP

Purpose Position at which the file is opened (temporary attribute).

Input Options are:

'$BOI'
Open at beginning-of-information (BOI).

'$ASIS'
Open without changing the file position.

'$EOI'
Open at end-of-information (EOI).

Output Integer as follows:

1 Open at beginning-of-information (BOI).

3 Open at end-of-information (EOI).

4 Open without changing the file position.

Default Open at beginning-of-information ('BOI').

Remarks If an existing file is opened for append access only, the only valid open position is EOI.

$OPEN_SHARE_MODES or $OSM

Purpose Set of access modes that can be granted to open requests within a job.

Input A string of one or more lists of keywords. The list has the form:

'(access mode), ... , *(access mode)*'

The access mode keywords are:

READA
Read access.

SHORTEN
Shorten access.

APPEND
Append access.

MODIFY
Modify permission (file statistics are kept).

EXECUTE
Execute access.

WRITE
Selects SHORTEN, APPEND, and MODIFY

ALL
Selects READ, SHORTEN, APPEND, MODIFY, and EXECUTE

NONE
No sharing by concurrent instances of open within the tasks.

Output A FORTRAN program should not fetch the $OPEN_SHARE_MODES value from the FIT. It is stored as an address which the program cannot use.

Default '(NONE)'

Remarks

- To set the share mode for a file, you must store the value in the FIT before the file is opened because the value only takes effect for the next instance of open.
- This FIT value is used by the FILEIS, FILEDA, and STOREF calls if the associated file is not open.
- When you specify this FIT value, it replaces any prior specification of $OPEN_SHARE_MODES.
- This FIT value controls sharing of a file within a job. When specified by the first of several concurrent instances of open, this option constrains the access modes of subsequent concurrent instances of open.

- If you specify more than one value, the first one is considered the preferred value and the remaining values are considered alternatives. When there are no outstanding instances of open, the preferred value is used to constrain the modes of access granted to subsequent concurrent instances of open. The alternatives are used only when there is another outstanding instance of open and the preferred value does not include the modes of access specified by other concurrent instances of open.

$PASSWORD or $P

Purpose Password that is compared with the password of an existing file. If the file is being created, the password is assigned to the file.

Input 1- to 31-character string indicating the password.

Output A FORTRAN program should not fetch the $PASSWORD value from the FIT. It is stored as an address which the program cannot use.

Default None

Remarks You must use the password on some NOS/VE commands, such as ATTACH_FILE and DELETE_FILE.

$PRIMARY_KEY_ADDRESS or $PKA

Purpose Location to which the primary-key value is returned.

Input Variable name.

NOTE

The primary-key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the $PRIMARY_KEY_ADDRESS value from the FIT. It is stored as an address which the program cannot use.

Default 0 (the primary-key value is not returned).

Remarks If the $PRIMARY_KEY_ADDRESS value in the FIT is nonzero, get calls issued while an alternate key is selected return the primary-key value of the record read to the specified location.

RL (Record Length)

Purpose Either the number of bytes written by a PUT call or the number of bytes read by the last GET or GETN call (parameter).

Input Integer from 1 through the maximum record length.

Output A FORTRAN program should not fetch the RL value from the FIT. It is stored as an address which the program cannot use.

Default When writing a variable-length (U or V) record, the record length must be specified. When writing a fixed-length (F) record, the maximum record length is used as the record length value.

$RECORD_LIMIT or $RL

Purpose Maximum number of records in the file (preserved attribute).

Input Integer from 1 through 4398046511103 ([2**42] - 1).

Output A FORTRAN program should not fetch the $RECORD_LIMIT value from the FIT. It is stored as an address which the program cannot use.

Default 4398046511103 ([2**42] - 1)

Remarks After the file is first opened, the RECORD_LIMIT attribute value is stored with the file. However, you can change the RECORD_LIMIT attribute value of an existing file with the NOS/VE command CHANGE_FILE_ATTRIBUTES.

$RECORD_TYPE or $RT

Purpose Record type (preserved attribute).

Input Options are:

'VARIABLE' or 'V'

Variable-length records.

'FIXED' or 'F'

Fixed-length records.

'UNDEFINED' or 'U'

Undefined-length records.

'TRAILING_CHARACTER_DELIMITED' or 'TRAILING' or 'TCD' or 'T'

Trailing_character_delimited records.

Output Integer as follows:

0 Variable-length (V) records.

1 Fixed-length (F) records.

7 Undefined-length (U) records.

Default Undefined-length (U) records.

Remarks

- The keyed-file interface processes record types U and V the same.
- The keyed-file interface does not support the trailing_character_delimited record type.

$RECORDS_PER_BLOCK or $RPB

Purpose Estimated number of records to be stored in each data block of a new file.

NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.

Input Integer from 1 through 65535.

Output A FORTRAN program should not fetch the $RECORDS_PER_BLOCK value from the FIT. It is stored as an address which the program cannot use.

Default Two records per block.

$SKIP_COUNT or $SC

Purpose Either the number of records to be skipped by the next SKIP call (parameter) or the residual skip count from the last SKIP call.

Input Integer. If the skip count is positive, a SKIP call skips forward the specified number of records. If the skip count is negative, a SKIP call skips backward the specified number of records.

Output A zero skip count indicates that the skip operation completed. A nonzero value indicates that the skip operation did not complete.

A nonzero value returned is the residual skip count. A residual skip count is the difference between the requested skip count and the actual number of records skipped.

Default 0 (no file repositioning).

Remarks The returned skip count is nonzero when the SKIP call encounters the BOI or EOI of the file before it completes the skip. To determine the file position, call IFETCH to return the $FILE_POSITION value.

$WAIT_FOR_ATTACHMENT or $WFA

Purpose Number of milliseconds to wait for the attachment of a file.

Input Integer

Output A FORTRAN program should not fetch the $WAIT_FOR_ATTACHMENT value from the FIT. It is stored as an address which the program cannot use.

Default 0 (Do not wait)

Remarks

- If you do not specify a wait time, the abnormal status PFE$CYCLE_BUSY is returned if the requested file cycle is busy. A file cycle is busy if the attach request specifies a usage selection set or share selections set that is incompatible with the current attaches of the file.
- If you specify a wait time for a file that is currently busy, other attach requests for the file cycle can be processed while your task waits for the file.
- This FIT value is used by the FILEIS, FILEDA, and STOREF calls if the associated file is not open.

$WAIT_FOR_LOCK or $WFL

Purpose Indicates whether the lock request should wait until the lock is available or the time limit has been reached.

Input Options are:

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'

The request waits for the lock.

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'

The request does not wait for the lock.

Output Integer as follows:

-1 The request waits for the lock (YES).

0 The request does not wait for the lock (NO).

Default YES (the request waits for the lock).

Remarks

- This FIT value may be used by GET, GETN, LOCKF, and LOCKK calls as follows:
  - Used by GET and GETN calls when the FIT value $GET_AND_LOCK is YES (-1).
  - Used by LOCKF and LOCKK if the wfl parameter is omitted from the call.
- When waiting is requested, the call checks for a possible deadlock. If a deadlock exists with another task, it immediately returns a nonfatal error status.
- If the lock is owned by another instance-of-open of the same task, a self-deadlock exists and the call immediately returns a nonfatal error status.
- You can change the maximum waiting period for the lock (used if wfl is YES). The default value is 60 seconds. To change the waiting period, create an NOS/VE integer variable named AAV$RESOLVE_TIME_LIMIT and initialize it to the new waiting period value in seconds. For example, this call executes an NOS/VE command that sets the waiting period at 45 seconds:

```
CALL SCLCMD ('var AAV$RESOLVE_TIME_LIMIT_: integer=45;varend')
```

  Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks, specify the scope as JOB within the VAR VAREND structure.

$WORKING_STORAGE_ADDRESS or $WSA

Purpose Location to which data is read and from which data is written (parameter).

Input Variable name.

NOTE

The working storage area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the $WORKING_STORAGE_ADDRESS value from the FIT. It is stored as an address which the program cannot use.

Remarks

- You can specify the $WORKING_STORAGE_ADDRESS location either on a STOREF call or on a get or put call. When you specify a $WORKING_STORAGE_ADDRESS location on a call, the $WORKING_STORAGE_ADDRESS location is stored in the FIT and used by all subsequent get or put calls until another $WORKING_STORAGE_ADDRESS location is specified.
- The length of the working-storage area is stored in the FIT as the $WORKING_STORAGE_LENGTH value.

$WORKING_STORAGE_LENGTH or $WSL

Purpose Length, in bytes, of the working storage area.

Input Integer less than or equal to the maximum record length value.

Output A FORTRAN program should not fetch the $WORKING_STORAGE_LENGTH value from the FIT. It is stored as an address which the program cannot use.

Default For read requests, the maximum record length value; for write requests, the record length value.

Remarks For read requests, $WORKING_STORAGE_LENGTH may be used to restrict the portion of a record that is read. When $WORKING_STORAGE_LENGTH is less than the actual record length, a portion of the record is read and a condition is returned to indicate a partial read.

Sort/Merge Interface 8

What Sort/Merge Does 8-1

Sort Keys 8-2
  Multiple Keys 8-2

Defining Sort Keys 8-3
  Key Length and Position 8-3
  Key Type 8-4
    Collating Sequences 8-5
    Numeric Data Formats 8-5
  Sort Order 8-9

Specifying the Record Length 8-10
  Short Records 8-10
    Exception Processing for Partial Numeric Key Sum Fields 8-11
    Exception Processing for Partial Sum Fields 8-11
  Zero-Length Records 8-12
  Invalid Records 8-13
    Exception Processing for Invalid Key Data 8-14
    Exception Processing for Summing Errors 8-14

Performance Considerations 8-15
  Limiting Memory Usage 8-15
  Page_Aging_Interval 8-16

Sort/Merge Procedure Calls 8-16.1
  Attaching the Sort/Merge Object Library 8-16.1
  Order of the Procedure Calls 8-16.1
  Characteristics of Sort/Merge Files 8-16.2
  Specifying Input and Output Files 8-16.2
  Owncode Procedures 8-16.2
  SM5CC 8-18
  SM5DUCT 8-19
  SM5E 8-20
  SM5EL 8-21
  SM5END 8-22
  SM5ENR 8-23
  SM5ERF 8-24
  SM5FMA 8-25
  SM5FROM 8-26
  SM5KEY 8-27
  SM5LCT 8-30
  SM5LIST 8-31
  SM5LO 8-32
  SM5MERG 8-33
  SM5OFL 8-35
  SM5OMIT 8-36
  SM5OMRL 8-37
  SM5OWNn 8-38
  SM5RETA 8-39
  SM5SEQA 8-40
  SM5SEQN 8-41
  SM5SEQR 8-42

SM5SEQS 8-43
SM5SORT 8-44
SM5ST 8-45
SM5SUM 8-46
SM5TMA 8-47
SM5TO 8-48
SM5VER 8-50
SM5ZLR 8-51

Owncode Procedures 8-52
Owncode Procedure Parameters 8-54
Owncode Procedure Record Length 8-55

Owncode 1: Processing Input Records 8-56
One or More Input Files Specified 8-56
Input Files Not Specified 8-57

Owncode 2: Processing Input Files 8-58
One or More Input Files Specified 8-58
Input Files Not Specified 8-58

Owncode 3: Processing Output Records 8-59
Output File Specified 8-59
Output File Not Specified 8-60

Owncode 4: Processing the Output File 8-61
Output File Specified 8-61
Output File Not Specified 8-61

Owncode 5: Processing Records With Equal Keys 8-62

Using FORTRAN Procedure Calls 8-64
Example 1: Sorting the Dean's List 8-64
Example 2: An In-Memory Sort 8-68

Creating an Object Library 8-70

Summing Records 8-72

Defining Your Own Collating Sequence 8-73

Sort/Merge Interface 8

Sort/Merge is a set of procedures that NOS/VE provides so you can sort and merge data. Sort/Merge can be used with NOS/VE commands, or with procedure calls from within a program written in COBOL, CYBIL, or FORTRAN.

This chapter introduces the functions and features of Sort/Merge using FORTRAN procedure calls.

What Sort/Merge Does

The purpose of sorting is to arrange items in order. The purpose of merging is to combine two or more sets of preordered items. Ordered information makes reports more meaningful and suggests critical relationships. Searches for information are faster with ordered lists.

The purpose of Sort/Merge is to arrange records in the sequence you specify. You describe the records you want to sort or merge and how Sort/Merge is to order them.

Sort/Merge can:

- Sort or merge records from as many as 100 files with one call to Sort/Merge.
- Sort character and noncharacter key types.
- Read input records with variable-length (V), fixed-length (F), or trailing-character-delimited (T) record type.
- Read input records from sequential, indexed-sequential, or direct-access files. It can write output records to sequential or indexed-sequential files.
- Read input records from and write output records to memory areas, mass storage files, and magnetic tape files.
- Sort according to 12 predefined collating sequences, 13 numeric formats, and one or more user-defined collating sequences.
- Sum fields in records that have equivalent key values.
- Use owncode procedures to insert, substitute, modify, or delete records during Sort/Merge processing.
- Be called from any language that matches the calling sequence although some restrictions may apply (described later).

Merge capabilities are more restricted than sort capabilities. Merge input records cannot be supplied by owncode procedures. Records to be merged must be presorted. Records to be merged and summed must be pre-sorted and pre-summed.

FORTRAN sorts are initiated with the SM5SORT procedure call and merges are initiated with the SM5MERG procedure call. You specify processing requirements for the sort or merge with various procedure calls.

Sort Keys

Sort or merge operations are based on the ordering of fields assigned to the data to be sorted or merged. These fields are called sort keys. This section discusses what sort keys are and how a key is defined.

A sort key is a field of data within each input record. Sort/Merge uses the contents of the sort key to determine the position of the record within the sorted sequence of records.

Data must be aligned correctly in a sort key field. Character data must be left-justified in the field, and numeric data must be right-justified in the field.

Multiple Keys

A file can be sorted on more than one sort key. The combined length of all key fields in a record cannot exceed 1023 bytes.

The first key you specify is the most important key and is called the major sort key. This key is sorted or merged first. The keys you specify after the first key are of lesser importance and are called minor sort keys. The minor keys are numbered in the order they are specified.

For example, if three sort keys are specified, the first key is the major sort key (key number 1), the next key listed is a minor key (key number 2), and the third key is another minor key (key number 3).

When two or more records have an equal major key, Sort/Merge determines the order by looking at the subsequent minor keys in the following order: key number 2, key number 3, and so on. Sort/Merge compares the minor keys until either an unequal key is found, or until there are no more keys.

For example, university student records could be sorted using multiple sort keys. Assume each record includes the last name and first and middle initials, the student number, the date of birth, the field of study, the grade point average, and a code representing class (freshman, sophomore, junior, senior); all the fields are written with character data. The file could be maintained with the student number as the major key since records are normally retrieved by specifying the student number. The file can be sorted by the name in alphabetic order when a list of student names is needed.

When a university department needs to know which students are majoring in fields within the department, the file can be sorted on the field of study. The same sort can specify the name as a minor key so that records with the same field of study are also sorted in alphabetic order by the name. The file can be sorted by the class code as the major key and by the grade point average in descending numeric order as a minor key. This would produce a list of students sorted by class code with the students having the highest grade point average at the beginning of the list.

Defining Sort Keys

You must describe to Sort/Merge every field of data that you want used as a sort key. Sort key descriptions include the following information:

- Starting location of the key within the record
- Key length
- Type of data in the key field
- Sort order

You can define sort keys with SM5KEY procedure calls. The options and assumed values for describing sort keys are discussed in the following paragraphs.

Key Length and Position

You define key field length and position by specifying the first byte of the field and either the number of bytes in the field (length of the field) or the last byte of the field. The leftmost byte in a record is counted as number 1. For character data, each character is 8 bits and occupies 1 byte. For example, if you want to specify the name of the university student file as a sort key, and the name field is the leftmost field in the record, you specify the first byte as 1. If the name field is 20 characters long, you specify the length as 20.

Sort/Merge interprets the integers you specify for key length and position as bit numbers when the key type (discussed later in this chapter) specifies bits; otherwise, byte numbers are assumed. The first bit is numbered 1. Table 8-1 lists the maximum key field lengths for each key type. Sort/Merge allows key fields to overlap other key fields, except for the following:

- Key fields that are ordered by collating sequences defined with the alter option cannot overlap other key fields.
- Key fields cannot overlap sum fields.

Table 8-1. Maximum Key Field Sizes

Key Type	Maximum Size (in Bytes)	Key Type	Maximum Size (in Bytes)
Character	1023	BINARY	8
NUMERIC_FS	1023	BINARY_BITS	8184 (bits)
NUMERIC_LO	38	INTEGER	8
NUMERIC_LS	38	INTEGER_BITS	8184 (bits)
NUMERIC_NS	38	PACKED	19
NUMERIC_TO	38	PACKED_NS	19
NUMERIC_TS	38	REAL	8 or 16

Key Type

You specify the type of data in a key field with the name of a collating sequence or with the name of a numeric data format. The data in a key field can be character or noncharacter. Character data is represented in the computer as ASCII code values. To indicate the key type for character data, you specify the name of a collating sequence; for numeric character data, you specify the name of a numeric data format. Noncharacter data is represented in the computer as binary values, in packed decimal format, or in floating-point format.

The difference between the internal representation of character and noncharacter data is shown in figure 8-1.

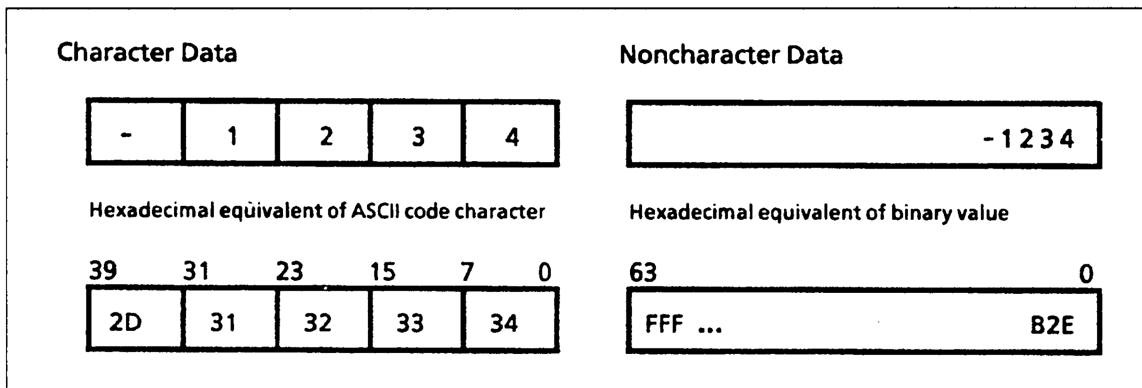


Figure 8-1. Internal Data Representation

If a sort key field contains any characters that are not meaningful for the key type you specify (an alphabetic character in a field defined as a numeric key, for example), the key field is considered to contain invalid data and so the record is invalid. The processing of invalid records is described later in this chapter.

The collating sequences and numeric data formats you can specify are discussed in the following paragraphs.

Collating Sequences

A collating sequence determines the precedence given to each character in relation to the other characters. You use a collating sequence for character data to determine the sort order. Character data must be in ASCII code characters.

Twelve predefined collating sequences are available to you as a Sort/Merge user. Six of the twelve predefined collating sequences are: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. If you do not specify a collating sequence, ASCII code is used. The predefined collating sequences are listed in appendix C, ASCII Character Set and Collating Weight Tables.

For Better Performance

Sort/Merge sorts fastest when using the ASCII collating sequence.

Numeric Data Formats

Numeric data can appear in a key field in one of the formats listed in table 8-2, Numeric Data Formats.

For Better Performance

For numeric data, the most efficient numeric data formats are INTEGER, BINARY, and REAL.

Except for the BINARY_BITS and INTEGER_BITS formats, each field must start and stop on character (byte) boundaries.

Numeric data can be signed or unsigned. For character numeric data that is signed, the sign can be a floating sign, an overpunch representation over the leading (leftmost) digit, a leading separate character, an overpunch representation over the trailing (rightmost) digit, or a trailing separate character.

Table 8-2. Numeric Data Formats

Name	Data Type	Sign	Comments
BINARY	Binary integer	None	The field starts and ends on character boundaries. Data is ordered according to numeric value.
BINARY_ BITS	Binary integer	None	The field does not start or end on character boundaries. Data is ordered according to numeric value.
INTEGER	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field starts and ends on character boundaries. Data is ordered according to numeric value.
INTEGER_ BITS	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field does not start or end on character boundaries. Data is ordered according to numeric value.
NUMERIC_ FS	Leading blanks, numeric characters	– sign for negative values; a + character is not allowed	The field contains leading blanks (leading zeros must be converted to blanks before calling Sort/Merge); if the value is negative, the rightmost leading blank must be converted to a minus sign. If the field contains no leading blanks or does not begin with a negative sign, the value must be positive. This format is equivalent to the FORTRAN I format, or the COBOL picture clause for zero suppressed editing of numeric item. Data is ordered according to numeric value.

(Continued)

Table 8-2. Numeric Data Formats *(Continued)*

Name	Data Type	Sign	Comments
NUMERIC_ LO	Numeric characters	Leading overpunch	All characters are decimal digits except the leading character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ LS	Numeric characters	Leading separate	All characters are decimal digits except the leading character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ NS	Numeric characters	None	All characters are decimal digits. Data is ordered according to numeric value.
NUMERIC_ TO	Numeric characters	Trailing overpunch	All characters are decimal digits except the trailing character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ TS	Numeric characters	Trailing separate	All characters are decimal digits except the trailing character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.

(Continued)

Table 8-2. Numeric Data Formats *(Continued)*

Name	Data Type	Sign	Comments
PACKED	Packed decimal	Signed	Data is ordered according to numeric value.
PACKED_NS	Unsigned packed decimal	Unsigned	Data is ordered according to numeric value. PACKED_NS is the same as COBOL COMPUTATIONAL–3 with no sign.
REAL	Normalized floating-point number, either single-precision (8 bytes) or double-precision (16 bytes)	Signed	All forms of zero are ordered equally. The order of indefinite values is undefined. The order of infinite values is ordered as if its value were infinity (can be signed infinity).

A floating sign is a negative sign embedded between leading blanks and the numeric characters. A floating sign can also be a negative sign followed by numeric characters. Leading zeros must be converted to blanks. Positive values in this format are not signed. The following examples are valid floating sign formats:

```
ΔΔ-1
ΔΔΔ1
ΔΔ-0
ΔΔΔ0
-123
1234
```

The following examples are invalid floating sign formats:

ΔΔ01	Leading zero not allowed
Δ–01	Leading zero not allowed
+123	Positive sign not allowed
ΔΔΔΔ	All-blank field not allowed

Diagnostic messages are issued for invalid floating sign formats or invalid overpunches.

A negative sign overpunch is equivalent to overstriking a digit with a – , which is a punch in row 11 of a punched card. A positive sign overpunch is equivalent to overstriking a digit with a + , which is a punch in row 12 of a punched card. When a signed overpunch digit is received as input, the digit is punched as indicated in the second column of table 8-3. When a signed overpunch digit is entered from a terminal or displayed as output, the digit appears as indicated in the third column of table 8-3. The hexadecimal value is in the fourth column.

Table 8-3. Sign Overpunch Representation

Sign and Digit	Input Punch	Input/Output Representation	Hexadecimal Value
+0	0	0	30
+1	1	1	31
+2	2	2	32
+3	3	3	33
+4	4	4	34
+5	5	5	35
+6	6	6	36
+7	7	7	37
+8	8	8	38
+9	9	9	39
+0	12–0	{	7B
+1	12–1	A	41
+2	12–2	B	42
+3	12–3	C	43
+4	12–4	D	44
+5	12–5	E	45
+6	12–6	F	46
+7	12–7	G	47
+8	12–8	H	48
+9	12–9	I	49
-0	11–0	}	7D
-1	11–1	J	4A
-2	11–2	K	4B
-3	11–3	L	4C
-4	11–4	M	4D
-5	11–5	N	4E
-6	11–6	O	4F
-7	11–7	P	50
-8	11–8	Q	51
-9	11–9	R	52
+0	12–8–4	<	3C
+0	12	&	26
-0	12–8–7	!	21
-0	11	–	2D

Sort Order

Sort/Merge can sort a key in ascending or descending order. If you do not specify a sort order, Sort/Merge sorts the key in ascending order.

When sorting a numeric key in ascending order, Sort/Merge sorts the key values in numeric order from least to greatest. When sorting a numeric key in descending order, Sort/Merge sorts the key values in numeric order from greatest to least.

A character key is sorted according to the collating sequence you specify for the key. When sorting a character key in descending order, Sort/Merge sorts the key values in reverse order of the collating sequence you specify.

Specifying the Record Length

Sort/Merge can sort records up to 65,535 bytes long. Sort/Merge determines the maximum and minimum record lengths for a file by its MAXIMUM_RECORD_LENGTH and MINIMUM_RECORD_LENGTH file attributes. The record length attributes are set when the file is created

The default sort key begins with the first byte in the record and extends to the smallest minimum record length value for all input files. If the minimum MINIMUM_RECORD_LENGTH attribute for all input files is 0, Sort/Merge uses 1 as the key length. If the minimum MINIMUM_RECORD_LENGTH attribute for all input files is greater than 1023, Sort/Merge uses 1023 as the key length.

When the Sort/Merge specification specifies an owncode 1 procedure and an owncode 3 procedure, but no input or output file, Sort/Merge expects all input records to be provided by the owncode 1 procedure and all output processing to be performed by the owncode 3 procedure. In this case you must specify the record length SM50FL or SM50MRL call.

Short Records

A short record is a record that does not contain all the key and sum fields defined for the sort or merge. Sort/Merge determines that a record is short when it reads the record from the input source.

NOTE

Records can become short when the system strips off all trailing blanks from variable-length (V) records. For example, when a variable-length (V) record containing all spaces is displayed by the NOS/VE command DISPLAY_FILE, the spaces are stripped from the record, leaving a zero-length record.

When Sort/Merge attempts to use a field in a record and finds that the field is entirely beyond the end of the record, it uses a default value for the field. For character keys, the default value is all spaces. For numeric keys and sum fields, the default value is zero in the appropriate format.

Sort/Merge uses the default value only when using the key value or the sum field value. It does not pass the default value to an owncode procedure or store it in the output record.

Sort/Merge processing differs when the field it attempts to use is only partially beyond the end of the record. If the partial field is a character key field, Sort/Merge pads it with spaces, but if the partial field is a numeric key field or a sum field, Sort/Merge processes it as an exception.

Exception Processing for Partial Numeric Key Sum Fields

Exception processing for partial numeric key fields is as follows:

1. The record is written to the exception records file if one is specified for the sort or merge.
2. If an exception records file exists, the record is removed from the sort or merge; otherwise, its order is left undefined.
3. The count of partial numeric key fields or sum fields is incremented. A warning error message gives the count at the end of the sort or merge.

Exception Processing for Partial Sum Fields

Exception processing for partial sum fields differs if an exception records file is specified:

1. If an exception records file is specified:
   a. Sort/Merge writes the record with the partial sum field to the exception records file. It writes the record with its original data as it was read from the input source.
   b. It then removes the record from the sort or merge.
2. If an exception records file is not specified:
   a. Sort/Merge keeps the record with the partial sum field in the sort or merge.
   b. Later, if Sort/Merge finds other records whose key values are equivalent to the record, it sums the records as if the partial sum field contains a valid value; it does not process the partial sum field as invalid data. However, because the results of summing with a partial field are undefined, the resulting contents of the sum field are undefined.

If Sort/Merge reads any records with partial sum fields, it returns a summary diagnostic at the end of the sort or merge, giving the number of records with partial sum fields.

Zero-Length Records

A zero-length record is a record that contains no data and so its record length is 0. The processing of zero-length records read from input files depends on the SM5ZLR call in the Sort/Merge specification.

By default, if the SM5ZLR call is omitted, Sort/Merge deletes all zero-length records from the sort or merge. This is the DELETE option.

However, instead of a DELETE specification, SM5ZLR can specify one of these processing options for zero-length records:

PAD

Assign default values to key fields and sum fields in zero-length records (as it would short records) and keep the zero-length records in the sort or merge.

LAST

Write all zero-length records at the end of the output file or memory area.

Zero-length records are never written to the exception records file if the DELETE option is selected. Zero-length records are written to the exception records file if the PAD option is selected and either of the following situations exist:

- If merge order verification is requested and the input files contain zero-length records which are not pre-sorted on the merge keys.
- If AMP$PUT_NEXT detects an error while writing a zero-length record. (In general, attempts to write zero-length records to an indexed-sequential file cause errors.)

If duplicate records are to be omitted (as specified by an SM5OMIT call) and the PAD option is specified for zero-length records, only one zero-length record is included in the sort or merge.

Zero-length records are passed to owncode procedures only if the PAD option is selected. When passing a zero-length record to an owncode procedure, Sort/Merge passes an empty array of the maximum record length and the record length parameter set to zero.

The counts kept in the result array for the sort or merge may differ depending on the SM5ZLR specification:

Word 2, number of records read

Zero-length records are always included in the count.

Word 6, number of records sorted or merged

Zero-length records are included only if the PAD option is selected.

Words 13, 14, and 15; number of records written, the minimum record length, and the average record length

Zero-length records are included in the computation of these values only if the PAD or LAST option is selected.

Word 17, the number of zero-length records deleted from the sort or merge

This count is kept only if the DELETE option is selected.

Invalid Records

Sort/Merge checks that all key fields contain data that is valid for the key type. It determines whether a sum field contains valid data only when it attempts to use the data. It does not validate any fields other than key fields and sum fields.

A record can also be determined to be invalid when it is written. Sort/Merge writes records to the output file using the system procedure AMP$PUT_NEXT. A record is considered invalid if AMP$PUT_NEXT returns an error when it attempts to write the record. For example, when writing an indexed-sequential file, AMP$PUT_NEXT returns an error if the primary-key value for the record is already in the file.

Invalid records are processed as exceptions. The processing performed depends on whether the invalid data is in a key field or a sum field.

Exception Processing for Invalid Key Data

A warning error is issued if a key field contains invalid data. The warning error results in the following actions:

1. The record is written to the exception records file if an exception records file was specified.
2. The record is deleted from the sort or merge if an exception file was specified. If an exception records file was not specified, the record remains in the sort or merge, but its place in the sort order is undefined.
3. A diagnostic message is issued, as controlled by the list options specification.
4. The sort or merge continues normally.

Exception Processing for Summing Errors

Sort/Merge detects summing errors only when it attempts to sum fields. Only one error is detected per sum field. The summing error is processed as an exception. If the LIST_OPTIONS requests detailed error reporting (DE), Sort/Merge issues a diagnostic for each summing error.

The exception processing performed for summing errors depends on the error detected and on whether an exception records file is specified for the sort or merge.

If an exception records file is specified:

1. Sort/Merge restores all sum fields of both records so their contents is the same as it was before summing of the two records began.
2. If the error is due to invalid data or an indefinite real, Sort/Merge knows that at least one of the sum fields in the record is in error; it does not know if the same sum fields in the other record is also in error. Therefore, it writes the record it knows to be in error to the exception records file and removes it from the sort or merge, but it leaves the other record in the sort or merge.
3. If Sort/Merge detects an arithmetic overflow or underflow error or finds that each record has invalid data in different sum fields, it knows that both records are in error. Therefore, it writes both records to the exception records file and removes both from the sort or merge.

If an exception records file is not specified:

1. Sort/Merge deletes one of the records. If one record is longer than the other, the shorter is deleted. Otherwise, either record could be deleted.
2. The other record remains in the sort or merge with undefined data in the sum field for which the error was detected. Summing is completed for the other sum fields.

Performance Considerations

To improve the performance of Sort/Merge in your programs, consider the following:

- Do not use owncode procedures except when necessary. Allow Sort/Merge to read the input records from files and write the output records to a file.
- Ensure that all key fields and sum fields are within the minimum record length for all input records. Additional processing is required for short records.
- If possible, use a fixed record length instead of a variable record length.
- Sort/Merge sorts fastest when using the ASCII collating sequence. For numeric data, the most efficient numeric data formats are INTEGER, BINARY, and REAL.
- Sort/Merge can read and write files faster if the files use the following default attributes:
  - Sequential file organization
  - F record type
  - System-specified blocking
  - No error-exit procedure
  - No file access procedure (FAP)
  - The padding character is space
- Use the optimum page_aging interval for your sort, as described under Page_Aging_Interval in this chapter.

Sort/Merge also executes faster when your site uses a larger page size.

Limiting Memory Usage

By default, Sort/Merge limits the memory assigned to its sorting array to 262,144 (256K) bytes. However, you can change this limit by defining a NOS/VE integer variable named SMV$MEMORY_USAGE_LIMIT. The integer you assign to the variable is used as the memory usage limit for subsequent sorts within the scope of the variable.

NOTE

The SMV$MEMORY_USAGE_LIMIT value is not used to specify limit memory usage for merges; it is used only for sorts (including the internal merge performed as part of a sort).

The integer assigned to the SMV$MEMORY_USAGE_LIMIT variable is the memory limit in 1024-byte (1K) units.

The minimum limit is 64. If you specify an integer less than 64, Sort/Merge uses the minimum limit of 64.

The maximum limit is 16,383. If you specify an integer greater than 16,383, Sort/Merge uses the maximum limit of 16,383.

A warning error is issued when you specify a value outside the range from 64 to 16,383.

Increasing the memory usage limit improves sort performance when:

- The file to be sorted is too large to be sorted in memory all at once (greater than 1,048,576 [1024K] bytes).
- The SMV$MEMORY_USAGE_LIMIT value is set to at least the size of the file to be sorted (up to 16,383K bytes). This allows the file to be sorted in memory all at once.
- The records in the file to be sorted are considerably disordered. If the records are already mostly sorted, increasing the memory usage limit has little effect on sort performance.

If you increase the memory usage limit for your sorts, you should also increase your page_aging_interval to prevent system thrashing. (The page_aging_interval job attribute is described in the next section.)

As an example of creating the variable, the SCL statement VAR/VAREND is used to create the SMV$MEMORY_USAGE_LIMIT variable and assign it the value 64.

```
call sclcmd ('var smv$memory_usage_limit: (local) integer = 64; varend')
```

Page_Aging_Interval

Page_aging_interval is the job attribute that controls how quickly pages are aged from the working set of a task. If you increase the memory usage limit for your sorts using the SMV$MEMORY_USAGE_LIMIT variable, you should also increase your page_aging_interval value.

The optimum page_aging_interval depends on the CYBER 180 model you use. A smaller value is appropriate for the faster models. For example, when the default memory usage limit of 256 pages is used, the optimum page_aging_interval for a CYBER 180/830 is about 500,000 microseconds, while, for a CYBER 180/860, the optimum value is about 100,000 microseconds.

To see your current page_aging_interval attribute value, enter the following NOS/VE command:

```
display_job_attribute, display_option=page_aging_interval
```

To change your page_aging_interval value, use the CHANGE_JOB_ATTRIBUTE command. For example, the following command changes the page_aging_interval to 500,000 microseconds:

```
change_job_attribute, page_aging_interval=500000
```

Sort/Merge Procedure Calls

FORTRAN Sort/Merge procedure calls must follow the same coding rules as other FORTRAN call statements. The Sort/Merge calls can be used by other languages that use the standard calling sequence.

Attaching the Sort/Merge Object Library

To use the Sort/Merge calls described in this chapter, you must add the following object library to the program library list before executing the program:

SMF$LIBRARY

The system attaches the file when you login, but you must add it to your library list so that modules can be loaded from the file.

For example, the following SET_PROGRAM_ATTRIBUTE command adds the object library to the program library list:

```
/set_program_attribute add_library=smf$library
```

Or, you can specify the object library on the EXECUTE_TASK command. This example specifies the object file LGO and the sort/merge object library SMF$LIBRARY:

```
/execute_task file=lgo library=smf$library
```

To read more about the program library list, see the NOS/VE Object Code Management Usage manual.

Order of the Procedure Calls

The procedures can be called in any order with two exceptions:

- SM5SORT or SM5MERG must be the first procedures called.
- SM5END must be the last procedure called.

Sort/Merge collects processing information until SM5END is called; the sort or merge is then performed.

Unless stated otherwise, a procedure can be called only once during a sort or merge.

Characteristics of Sort/Merge Files

Sort/Merge requires a value for the maximum record length for all procedure calls. You must specify a value for MAXRL on the SET_FILE_ATTRIBUTE command if the system default (MAXRL=256) is too small for your files.

By default, NOS/VE files have these characteristics:

- Block_type = system_specified (BT=SS)
- Record_type = variable (RT=V)

For files with other block types and record types, you must execute the SET_FILE_ATTRIBUTE command before the file is created and before the sort or merge.

Sort/Merge also attaches the input file or files with an acess mode of read and a share mode of read. To share the input file, other attaches to the input must also specify access and share modes of read.

For example, this shows opening the input file with access and share modes of read:

```
CALL SCLCMD('attach_file file=inpfile access_mode=read share_mode=read')
OPEN (unit=10, file='inpfile', err=100, end=120)
```

For more information on sharing files and access and share modes, see chapter 3, Sharing Keyed Files.

Specifying Input and Output Files

Input files are named by the SM5FROM procedure; output files are named by the SM5TO procedure. You can enter the Sort/Merge parameter and user-defined values in uppercase, lowercase, or a combination, because Sort/Merge treats lowercase letters as being equal to uppercase letters, except for owncode procedure names and collating sequences.

Owncode Procedures

If you specify an owncode procedure name in lowercase letters, Sort/Merge does not convert the name to uppercase letters unless you specify the true option on the SM5CC procedure.

Since some compilers convert procedure names to uppercase letters, you should specify owncode procedure names in all uppercase letters.

This page intentionally left blank.

SM5CC

Purpose

Specifies whether lowercase letters in owncode procedure names are to be converted to uppercase letters.

Format

CALL SM5CC(option)

Parameters

option

Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

Sort/Merge converts any lowercase letters in owncode procedure names to uppercase letters.

'FALSE', 'F', 'NO', 'N', 'OFF'

Sort/Merge does not convert lowercase letters in owncode procedure names.

If the SM5CC call is omitted, lowercase letters in owncode procedure names are not converted.

Remarks

- When Sort/Merge attempts to load an owncode procedure, it passes the procedure name as you have specified it on the SM5OWNn call. If you specify the name with lowercase letters, Sort/Merge passes the lowercase letters unless an SM5CC call requests conversion.
- The system stores entry point names using uppercase letters only. Therefore, if the loader is given a procedure name containing lowercase letters, it cannot find that name in the program library list and so it cannot to load the requested procedure.

Examples

This example calls SM5CC and specifies TRUE so owncode procedure names will be converted to uppercase letters:

```
CALL SM5CC('True')
```

SM5DUCT

Purpose Specifies a user-defined collation table.

Format **CALL SM5DUCT (key_type, collating_table_name)**

Parameters

key_type

Name you choose to call the collating sequence defined by the weight table. It is specified as the key type on the SM5KEY calls that use this collating sequence.

collating_table_name

Name of the 256-character string containing the collating weights.

Remarks

- Sort/Merge does not distinguish between lowercase and uppercase letters in the specified names.
- A Sort/Merge call sequence can include more than one SM5DUCT call.
- The total number of SM5SEQN, SM5LCT, and SM5DUCT calls in a Sort/Merge call sequence cannot exceed 100.
- The name SM5DUCT assigns to the collating sequence cannot be the name of a predefined collating sequence or another collating sequence already defined for the sort or merge.
- The weight table must already be loaded as part of the program. It must be a string declared by CHARACTER USER*256. Each character specifies the collating weight of the corresponding ASCII character.
- For more information, see the Collation Tables appendix in this manual.

SM5E

Purpose Specifies the file to which diagnostic messages for this sort or merge are written.

Format **CALL SM5E (file)**

Parameters **file**

Character expression specifying the file reference of the file. File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $NULL are assumed to be in the $LOCAL catalog.

If SM5E call is omitted, error messages are written to file $ERRORS.

Remarks

- Sort/Merge writes the error file only if it detects errors of at least the severity specified by the SM5EL call.
- Sort/Merge does not rewind the error file before or after it uses it.
- If you specify $NULL as the error file, diagnostic messages are not written.
- If you specify the same file as the listing file and as the error file (SM5E and SM5LIST), each diagnostic message is written only once to the file. (Otherwise, each message is written twice, once to the error file and once to the listing file.)
- The error level reported to the error file is specified by the SM5EL call.

SM5EL

Purpose Specifies the minimum severity level to be reported on the error file.

Format CALL SM5EL (severity_level)

Parameters **severity_level**

Options are:

'I' or 'i'

All informational, warning, fatal, and catastrophic errors.

'W' or 'w'

All warning, fatal, and catastrophic errors.

'F' or 'f'

All fatal and catastrophic errors.

'C' or 'c'

Only catastrophic errors.

'NONE' or 'none'

No errors are written to the error file.

If the SM5EL call is omitted, all diagnostics are reported regardless of severity.

Remarks The error file is specified by the SM5E call.

Examples This SM5EL call specifies that all warning, fatal, and catastrophic errors are reported to the error file:

```
CALL SM5EL ('w')
```

SM5END

Purpose Terminates a sort or merge specification and initiates Sort/Merge processing.

Format **CALL SM5END**

Remarks The SM5END call is required. It must be the last in the sequence of Sort/Merge calls.

SM5ENR

Purpose

Allows compatibility with NOS Sort/Merge 5. NOS/VE does not use the specified value.

Format

CALL SM5ENR (value)

Parameters

value

An integer value indicating the estimated number of records to be sorted. The value can be from 1 through 16,777,215.

SM5ERF

Purpose Specifies the file to which invalid records are written.

Format **CALL SM5ERF (file)**

Parameters **file**

Character expression specifying the file reference of the exception records file. File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $NULL are assumed to be in the $LOCAL catalog.

If the SM5ERF call is omitted, exception records are not removed from the sort or merge. The order of records with invalid keys is undefined. The contents of sum fields for which summing errors are detected is also undefined.

Remarks

- The exception records file cannot also be the output file or an input file. Its file organization must be sequential; it cannot be a keyed file.
- If you specify $NULL as the exception records file, each exception record is deleted as it is written to the file.
- All records written to the exception records file are deleted from the sort or merge.
- The records written to the exception records file include:
  - Records containing invalid key data.
  - Records containing invalid sum data if Sort/Merge attempts to sum the data.
  - Records that caused an arithmetic overflow or underflow when their sum fields were summed.
  - Short records in which Sort/Merge found a partial numeric key field or partial sum field.
  - Out-of-order merge input records if merge order checking was requested by an SM5VER call.
  - Records for which the system procedure AMP$PUT_NEXT returned an error when it attempted to write the record to the output file for the sort or merge.
- A summary of the records written to the exception records file is written to the errors file and to the list file.

SM5FMA

Purpose Specifies a memory area to be read as a source of input records.

Format **CALL SM5FMA (variable, 'FIXED', max_record_length, number_of_records)**

Parameters

variable

Name of the memory location at which Sort/Merge begins reading input records.

'FIXED'

String expression specifying that each input record read from the memory area is the fixed length specified by the third parameter on the call.

max_record_length

Integer giving the fixed record length in bytes. The maximum input record size is 65,536.

number_of_records

Integer giving the number of records Sort/Merge is to read from the memory area.

If the SM5FMA call is omitted, all input records are read from files or supplied by owncode procedures.

Remarks

- A Sort/Merge specification can specify up to 100 sources of input records. These sources can be files or memory area; the sources are read in the order you specify them. Files are specified by SM5FROM calls; memory areas are specified by SM5FMA calls.
- When a memory area is used as an input record source, a sort cannot use an owncode 1 or owncode 2 procedure.
- The record order is undefined when a memory area specified by an SM5FMA call overlaps the memory area specified by the SM5TMA call.
- For an example of using SM5FMA in an in-memory sort, see Using FORTRAN Procedure Calls later in this chapter.

SM5FROM

Purpose Specifies one or more files from which input records are read.

Format **CALL SM5FROM (file,** ..., *file*)

Parameters **file**

Character expression specifying the file reference of an input file. The files are read in the order specified on the call. File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $NULL are assumed to be in the $LOCAL catalog.

If the SM5FMA call is omitted, input records are read from the specified memory area. Or, if SM5OWN1 is called, input records could be supplied by the owncode 1 procedure. Otherwise, Sort/Merge attempts to open and read file $LOCAL.OLD as the source of input records.

Remarks

- A Sort/Merge specification can specify up to 100 sources of input records. These sources can be files or memory areas; the sources are read in the order you specify them. Files are specified by SM5FROM calls; memory areas are specified by SM5FMA calls.
- All instances of open of the input files must be closed before the sort or merge begins. Sort/Merge opens each file before it reads it and closes it when it has finished reading it.
- Sort/Merge does not read past an end-of-partition delimiter embedded in an input file.
- The input files for a merge must be pre-sorted on the same keys used for the merge. For a merge with summing, the input files must also be pre-summed using the same sum fields specified for the merge.
- A Sort/Merge input file can reside on either mass storage or magnetic tape.
- The Sort/Merge output file can have sequential, direct-access, or indexed-sequential file organization and its record type can be variable (V), fixed-length (F), or trailing-character-delimited (T).

SM5KEY

Purpose Specifies a key field to be used by the sort or merge.

Format **CALL SM5KEY (first, length, key_type, order)**

Parameters

first

Integer expression specifying the first position of the key field. Bit positions are used for the BINARY_BITS and INTEGER_BITS key types, byte positions for all others. Positions are numbered from the left beginning with 1.

length

Integer expression specifying the number of positions in the key field. The number of bits are given for the BINARY_BITS and INTEGER_BITS key types, byte positions for all others.

To see the maximum key field sizes, see table 8-1.

key_type

Character expression specifying the numeric data format or, for character data, the collating sequence.

You can define a collating sequence name with an SM5DUCT, SM5LCT, or SM5SEQN call or use a predefined collating sequence. Options are:

'ASCII'

ASCII collating sequence.

'ASCII6'

OSV$ASCII6_FOLDED collating sequence.

'COBOL6'

OSV$COBOL6_FOLDED collating sequence.

'DISPLAY'

OSV$DISPLAY64_FOLDED collating sequence.

'EBCDIC'

OSV$EBCDIC collating sequence.

'EBCDIC6'

OSV$EBCDIC6_FOLDED collating sequence.

Appendix C, ASCII Character Set and Collating Weight Tables, lists the predefined collating sequence.

The following are the available numeric data formats:

'BINARY'

Binary integer starting and ending on byte boundaries.

'BINARY_BITS'

Binary integer not required to start or end on byte boundaries.

'INTEGER'

Two's complement binary integer starting and ending on byte boundaries.

'INTEGER_BITS'

Two's complement binary integer not required to start or end on byte boundaries.

'NUMERIC_FS'

Numeric characters with floating sign (FORTRAN I format or COBOL zero-suppressed editing item).

'NUMERIC_LO'

Numeric characters with leading overpunch sign.

'NUMERIC_LS'

Numeric characters with leading separate sign.

'NUMERIC_NS'

Numeric characters with no sign.

'NUMERIC_TO'

Numeric characters with trailing overpunch sign.

'NUMERIC_TS'

Numeric characters with trailing separate sign.

'PACKED'

Signed packed decimal.

'PACKED_NS'

Unsigned packed decimal.

'REAL'

Normalized floating-point number, single-precision (8 bytes) or double-precision (16 bytes).

order

Character expression specifying the order of the sort or merge operation. Options are:

'A' or 'a'

Ascending order

'D' or 'd'

Descending order

If the SM5KEY call is omitted, the only key field used begins at position 1 and extends through the smallest minimum record length of the input sources. However, the minimum key length used is 1 and the maximum key length used is 1023.

The key is sorted by the ASCII collating sequence in ascending order.

Remarks

- Sort/Merge treats lowercase letters in parameter values as being equal to uppercase letters.
- The combined length of all key fields defined for a sort or merge cannot exceed 1023 bytes.
- The total number of SM5KEY calls in a Sort/Merge call sequence cannot exceed 106.
- The significance of multiple keys corresponds to the order in which the keys are defined.
- Sort key fields can overlap other sort key fields with the following exceptions:
  - Key fields that are ordered by collating sequences defined with an SM5SEQA call cannot overlap other key fields.
  - Key fields cannot overlap sum fields.
- For more information, see the description of Short Records and Zero-Length Records earlier in this chapter.

SM5LCT

Purpose

Loads a collation table, that is, a weight table that defines a collating sequence. The table may be a NOS/VE predefined collation table or a user-defined collation table in an object library.

Format

CALL SM5LCT (key_type, collation_table_name)

Parameters

key_type

Name you choose to call the collating sequence defined by the weight table. It is specified as the key type on the SM5KEY calls that use this collating sequence.

The name cannot be the name of a predefined collating sequence or the name of a collating sequence you have already defined.

collation_table_name

Name of a predefined weight table or an object library module defining a collating sequence.

Remarks

- Sort/Merge treats lowercase letters as being equal to uppercase letters.
- The total number of SM5DUCT, SM5LCT, and SM5SEQN calls in a Sort/Merge specification cannot exceed 100.
- The weight table must be loadable by PMP$LOAD and have 256 weight values.
- For more information, see the collation table in Appendix C, ASCII Character Set and Collating Weight Tables.

SM5LIST

Purpose Specifies the name of the list file.

Format CALL SM5LIST (file)

Parameters file

Character expression specifying the file reference of the listing information file. File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $NULL are assumed to be in the $LOCAL catalog.

If the SM5LIST call is omitted, the default list file is $LIST.

Remarks

- Listing information includes the Sort/Merge version and level numbers, time and date, diagnostics, and statistics such as the number of records sorted or merged.
- If you specify the same file as the listing file and as the error file (SM5E and SM5LIST), each diagnostic message is written only once to the file. Otherwise, each message is written twice, once to the error file and once to the listing file.

SM5LO

Purpose Specifies the information written to the listing file.

Format **CALL SM5LO (option)**

Parameters **option**

Options are:

'OFF'

All listing information is suppressed.

'NONE'

Same as OFF keyword.

'DE'

Detailed exception information (valid only if SM5ERF is called).

'RS'

Record statistics for those records sorted or merged.

'MS'

Merge statistics for the records merged.

Remarks

- The minimum information Sort/Merge writes to the listing file is the page heading, error messages, the exception file summary, and the number of records sorted or merged.
- You can specify only one option with each SM5LO call, but the Sort/Merge specification can include more than one SM5LO call.

SM5MERG

Purpose

Signals the beginning of a sequence of Sort/Merge calls for a merge operation.

Format

CALL SM5MERG (array)

Parameters

array

Name of a one-dimensional array of 1 through 18 integers in which Sort/Merge returns statistics about the merge. Or, if you specify 0, Sort/Merge returns no statistics.

NOTE

The specified result array should be declared inside a common block. FORTRAN optimization requires that variables specified on a call, but modified after return from the call, occur only in common blocks.

Remarks

- SM5MERG must be the first procedure called for a merge operation.
- In the first word of the array, you must specify the number of values (0 through 17) you want returned. Values are returned in words 2 through 18. The array must be long enough to contain the number of values you request in the first word.
- The result array format is listed in table 8-4.

Table 8-4. Result Array Format

Array Element	Contents
1	Number of elements of results you want returned in the array (0 through 17)
2	Number of records read from input files or memory areas
3	Number of records deleted by an owncode 1 procedure
4	Number of records inserted by an owncode 1 procedure
5	Number of records inserted by an owncode 2 procedure
6	Number of records sorted or merged. The count does not include records written to the exception records file or zero-length records (unless the SM5ZLR call selects the PAD option.)
7	Number of records deleted by an owncode 3 procedure
8	Number of records inserted by an owncode 3 procedure
9	Number of records inserted by an owncode 4 procedure
10	Number of records written to the exception file
11	Number of records deleted by an owncode 5 procedure
12	Number of records combined by summing
13	Number of records written to the output file or memory area
14	Actual minimum record length of all input records
15	Average record length (total record length divided by the total number of input records)
16	Actual maximum record length of all input records
17	Number of zero-length records removed from the sort or merge because the default SM5ZLR option (DELETE) is selected.
18	Number of records with equivalent key values (duplicates) removed from the sort or merge as requested by an SM5OMIT call.

SM5OFL

Purpose Specifies the length of each fixed-length record entering the sort or merge from an owncode procedure.

Format CALL SM5OFL (fixed_length)

Parameters **fixed_length**

Integer expression specifying the fixed length in bytes. Valid values are from 1 through 65,535.

If SM5OWN1 and SM5OWN3 are called, but SM5FROM and SM5TO are not, an SM5OFL or SM5OMRL call is required. Otherwise, if SM5OFL and SM5OMRL are omitted, the record length is the largest MAXIMUM_RECORD_LENGTH attribute for the input and output files used by the sort.

Remarks

- A fatal error occurs if a owncode procedure supplies a record of any other length.
- You cannot call both SM5OFL and SM5OMRL for the same sort operation.

SM5OMIT

Purpose Specifies whether Sort/Merge outputs only one record in each set of records with equivalent key values.

Format **SM5OMIT (option)**

Parameters **option**

Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

Duplicates are omitted.

'FALSE', 'F', 'NO', 'N', 'OFF'

Duplicates are not omitted.

If the SM5OMIT call is omitted, duplicates are not omitted. The processing of records with equivalent key values depends on whether SM5OWN5, SM5RETA, or SM5SUM is called. If all of these calls are omitted, records with equivalent key values remain in the sort or merge, but their relative order is undefined.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- When duplicates are omitted, Sort/Merge removes the shorter duplicate records from the sort or merge. When the duplicates have the same length, any of the duplicates could be the one that is kept.
- A count is kept in word 18 of the result array of the number of duplicate records deleted from the sort or merge due to an SM5OMIT call. (The result array is specified on the SM5MERG or SM5SORT call.)
- Duplicates omitted by an SM5OMIT call are not written to the exception records file.
- Zero-length records are processed as duplicates only if the SM5ZLR call specifies the PAD option.

SM5OMRL

Purpose

Specifies the maximum length of any record entering the sort or merge from an owncode procedure.

Format

CALL SM5OMRL (maximum_length)

Parameters

maximum_length

Integer expression specifying the maximum length in bytes.

If SM5OWN1 and SM5OWN3 are called, but SM5FROM and SM5TO are not, an SM5OFL or SM5OMRL call is required. Otherwise, if SM5OFL and SM5OMRL are omitted, the record length is the largest MAXIMUM_RECORD_LENGTH attribute for the input and output files used by the sort.

Remarks

- SM5OMRL need not be called if Sort/Merge has an input or output file with a maximum record length at least as long as the maximum record length of the user-supplied records.

- You cannot call both the SM5OFL and SM5OMRL procedures for the same sort operation. If all records supplied by owncode procedures have the same length, SM5OFL should be called instead of SM5OMRL.

SM5OWNn

Purpose Specifies a user-written (owncode) procedure to be executed each time a certain event occurs during the sort or merge.

Format

CALL SM5OWN1(name)	Specifies the name of the owncode 1 procedure executed each time a sort reads an input record.
CALL SM5OWN2(name)	Specifies the name of the owncode 2 procedure executed each time a sort finishes reading an input file.
CALL SM5OWN3(name)	Specifies the name of the owncode 3 procedure executed each time a sort or merge is ready to write an output record.
CALL SM5OWN4(name)	Specifies the name of the owncode 4 procedure executed each time a sort or merge finishes writing its output records.
CALL SM5OWN5(name)	Specifies the name of the owncode 5 procedure executed each time a sort or merge finds two records with equivalent key values.

Parameters **name**

Character expression specifying the name of an owncode procedure.

The name must be specified using all uppercase letters unless the sort or merge calls SM5CC with the true option.

Owncode procedures are executed only if they are specified.

Remarks

- Merge specifications cannot call SM5OWN1 or SM5OWN2.
- Sort/Merge specifications that call SM5FMA cannot call SM5OWN1 or SM5OWN2. Sort/Merge specifications that call SM5TMA cannot call SM5OWN3 or SM5OWN4.
- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- For further information about owncode procedures, see the discussion later in this chapter.

SM5RETA

Purpose Specifies whether input records having equal keys are to be output in the same order they are input.

Format **CALL SM5RETA (option)**

Parameters **option**

Options are:

'YES'

Records with equal keys retain their original order.

'NO'

Records with equal keys may not retain their original order.

If this argument is omitted (no argument list specified), the default is YES.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- If you select the 'YES' option and specify more than one input source, the order in which you specify the input sources is the order in which records with equal keys will be written.
- Maintaining the original order of records with equal key values increases the required processing time because Sort/Merge must keep track of the input order.

SM5SEQA

Purpose Used with the SM5SEQS call to specify whether characters are altered in the output. If characters are altered, all characters in the value step specified by the preceding SM5SEQS call are output as the first character in the value step.

Format **CALL SM5SEQA (option)**

Parameters **option**

Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

Alters the equated characters.

'FALSE', 'F', 'NO', 'N', 'OFF'

Does not alter the equated characters.

If the SM5SEQA call is omitted, characters are not altered.

Remarks SM5SEQA is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQS, and SM5SEQR.

Examples The sequence of calls below converts all commas and semicolons to spaces:

```
CALL SM5SEQN ('ALTERSQ')
CALL SM5SEQS (' ', ',', ';')
CALL SM5SEQA ('YES')
```

SM5SEQN

Purpose

Specifies the name of the collating sequence specified by the following SM5SEQS, SM5SEQR, and SM5SEQA calls.

Format

CALL SM5SEQN (name)

Parameters

name

Character expression specifying the name of the user-defined collating sequence.

Remarks

- The end of the collating sequence definition is indicated by any statement other than an SM5SEQS, SM5SEQR, and SM5SEQA call.
- The specified name cannot be the same as that of any predefined collating sequence or user-defined collating sequence that you have already defined for the sort or merge.
- The specified name is used as the key type on SM5KEY calls defining key fields to be ordered by the user-defined collating sequence.

Examples

This statement names a user-defined collating sequence:

```
CALL SM5SEQN ('MYSEQ')
```

This statement defines a key field that uses the user-defined collating sequence:

```
CALL SM5KEY(1, 10, 'MYSEQ', 'A')
```

SM5SEQR

Purpose

Defines the position of the remainder value step in the collating sequence being defined. The remainder value step consists of all characters that have not been included in value steps defined by SM5SEQS calls.

Format

CALL SM5SEQR (option)

Parameters

option

Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

The remainder value step is defined at this position.

'FALSE', 'F', 'NO', 'N', 'OFF'

The remainder value step is not defined.

If the SM5SEQR call is omitted, the last value step in the collating sequence is defined as the remainder value step.

Remarks

SM5SEQR is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQS, and SM5SEQA.

Examples

The sequence below defines a collating sequence with two value steps: all nondigits followed by all digits.

```
CALL SM5SEQN ('DIGITS')
CALL SM5SEQR ('YES')
CALL SM5SEQS ('0','1','2','3','4','5','6','7','8','9')
```

SM5SEQS

Purpose

Specifies a value step in the collating sequence being defined.

A value step consists of one or more characters that are to have the same collating weight in the sequence.

The first CALL SM5SEQS statement specifies the first value step, the second SM5SEQS statement specifies the second value step, and so on until the collating sequence is completely defined.

Format

CALL SM5SEQS (char, ..., *char*)

Parameters

char

Character expression specifying a character in the value step.

Remarks

SM5SEQS is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQR, and SM5SEQA.

Examples

- This statement defines a value step consisting of one character:

```
CALL SM5SEQS ('A')
```

- This statement defines a value step consisting of several characters:

```
CALL SM5SEQS ('1', '2', '3', '4')
```

SM5SORT

Purpose Signals the beginning of a sequence of Sort/Merge calls for a sort operation.

Format **CALL SM5SORT (array)**

Parameters **array**

Name of an integer array in which Sort/Merge returns statistics about the merge. Or, if you specify 0, Sort/Merge returns no statistics.

NOTE

The specified result array should be declared inside a common block. FORTRAN optimization requires that variables specified on a call, but modified after return from the call, occur only in common blocks.

Remarks

- SM5SORT must be the first procedure called for a sort operation.
- In the first word of the array, you must specify the number of values (0 through 17) you want returned. Values are returned in words 2 through 18. The array must be long enough to contain the number of values you request in the first word.
- To see the result array format, see table 8-4.

SM5ST

Purpose Returns the severity level of the most severe error encountered during the sort or merge operation.

Format **CALL SM5ST (severity_level)**

Parameters **severity_level**

Variable in which Sort/Merge returns an integer indicating the highest severity level of all errors detected during the sort or merge:

0	No errors
10	Informational errors
20	Warning errors
30	Fatal errors
40	Catastrophic errors

SM5SUM

Purpose Specifies that summing is to be performed on the specified fields.

Format **CALL SM5SUM (first, length, type, repeat)**

Parameters

first

Integer expression specifying the first byte or bit of the sum field (numbered from the left starting with 1).

length

Integer expression specifying the number of bytes or bits in the sum field.

type

Character expression specifying the numeric data format. The numeric data formats are listed in table 8-2.

repeat

Integer greater than zero specifying the number of times the field repeats in the record.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- Sum fields cannot overlap one another. Sum fields cannot overlap key fields.
- SM5SUM can be called up to 100 times for each sort or merge.
- If SM5SUM is called, Sort/Merge processes records with equivalent values by combining the records into one output record. The sum fields contain the sums of the values in the corresponding sum fields in the input records. The rest of the record is taken from the longest of the original input records.
- To read about exception processing for partial sum fields, see the discussion under short records in this chapter.

SM5TMA

Purpose Specifies a memory area to used as the destination of output records.

Format **CALL SM5TMA (variable, 'FIXED', max_record_length)**

Parameters

variable

Name of the memory location at which Sort/Merge begins writing output records.

'FIXED'

String expression specifying that each input record written to the memory area is the fixed length specified by the third parameter on the call.

max_record_length

Integer giving the fixed record length in bytes. The maximum input record size is 65,536.

If the SM5TMA call is omitted, all output records are written to an output file or processed by an owncode 3 procedure.

Remarks

- A Sort/Merge specification can specify only one destination for output records. The destination can be a file or a memory area, but not both. A file is specified by an SM5TO call; a memory area is specified by an SM5TMA call.
- When a memory area is used as the destination for output records, the sort or merge cannot use owncode 3 or owncode 4 procedures.
- The record order is undefined when a memory area specified by an SM5FMA call overlaps the memory area specified by the SM5TMA call.
- A count of the records written to the memory area is kept in word 13 of the result array. (The result array is specified on the SM5SORT or SM5MERG call.)
- For an example of using SM5TMA in an in-memory sort, see Using FORTRAN Procedure Calls later in this chapter.

SM5TO

Purpose Specifies the file to receive the sorted or merged output records.

Format **CALL SM5TO (file)**

Parameters **file**

Character expression specifying the file reference of the file. File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $NULL are assumed to be in the $LOCAL catalog.

If the SM5TMA call is omitted, output records are written to the specified memory area. Or, if SM5OWN3 is called, output records are processed by an owncode 3 procedure. Otherwise, Sort/Merge writes the output records to file $LOCAL.NEW.

Remarks

- The output file cannot also be an input file or the exception records file or the error file or the list file.
- The file must be closed when the sort or merge begins. Sort/Merge closes the file when it completes the sort or merge.
- The Sort/Merge output file can reside on either mass storage or magnetic tape.
- The Sort/Merge output file can have either sequential or indexed-sequential file organization and its record type can be variable (V), fixed-length (F), or trailing-character-delimited (T).
- The Sort/Merge output file cannot use the direct-access file organization.
- If the output file is an indexed-sequential file with a nonembedded primary key, the primary-key value is removed from the beginning of the record when it is written to the output file.

  The removed primary-key value is stored in the primary index of the file. The record data stored is shortened by key_length characters.
- If the output file is an indexed-sequential file, the major sort key must be the primary key defined for the output file.

  The indexed-sequential file organization requires that each primary-key value be unique. Therefore, the value in the major sort key field must be unique for each output record. This can be ensured by specifying the OMIT_DUPLICATES=YES parameter or using an owncode 5 procedure.

- If the output (TO) file is an indexed-sequential file, Sort/Merge checks the KEY_POSITION, KEY_LENGTH, and KEY_TYPE attributes:
  - If the major sort key position does not match the KEY_POSITION attribute value, Sort/Merge issues a fatal error and terminates.
  - If the major sort key length does not match the KEY_LENGTH attribute value, Sort/Merge issues a warning error and changes the major sort key length to match the primary key length.
  - If the major sort key type does not match the KEY_TYPE attribute value, Sort/Merge issues a warning error and changes the major sort key type if the KEY_TYPE value is UNCOLLATED or INTEGER. (It does not issue a warning or change the key type if the KEY_TYPE value is COLLATED.)
    - If the KEY_TYPE is UNCOLLATED, the major sort key type is changed to ASCII.
    - If the KEY_TYPE is INTEGER, the major sort key type is changed to INTEGER.

SM5VER

Purpose Specifies whether Sort/Merge checks that the input records to a merge are in sorted order.

Format **CALL SM5VER (option)**

Parameters **option**

Options are:

'TRUE', 'T', 'YES', 'Y', 'ON'

The order of merge input records is verified.

'FALSE', 'F', 'NO', 'N', 'OFF'

The order of merge input records is not verified.

If the SM5VER call is omitted, the order of merge input records is not verified. Out-of-order input records remain in the merge. Their order in the output file is undefined.

Remarks

- If merge order verification is requested and Sort/Merge finds an input record out of order, it issues a warning message.

  If an exception records file has been specified (SM5ERF), any out-of-order input records are written to the exception records file and then deleted from the merge.

- If you include an SM5VER call is a sort specification, Sort/Merge issues a warning message, but otherwise ignores the call.

SM5ZLR

Purpose

Specifies the disposition of zero-length records.

NOTE

The SM5ZLR option applies only to records read from input files; it does not apply to records read from memory areas or supplied by owncode procedures.

Format

CALL SM5ZLR (keyword)

Parameters

keyword

Options are:

'DELETE'

Each zero-length record is deleted from the sort or merge. (The deleted records are not written to the exception records file.)

'PAD'

Each zero-length record is processed as a short record. Key fields are assigned default values (spaces for character keys; zero for numeric keys).

'LAST'

Each zero-length record is written at the end of the output.

If the SM5ZLR call is omitted, each zero-length record is deleted from the sort or merge.

Remarks

For more information about zero-length records, see the discussion earlier in this chapter.

Owncode Procedures

You can write subprograms to insert, substitute, modify, or delete input and output records during Sort/Merge processing. Such a subprogram, called an owncode procedure, is executed each time the sort or merge reaches a certain point in Sort/Merge processing. Figure 8-2 illustrates the points at which Sort/Merge can call owncode procedures.

Sort/Merge passes a record to the owncode procedure, which processes the record. When the record is returned to Sort/Merge from the owncode procedure, Sort/Merge processes the record according to a code passed by the owncode procedure.

Owncode procedures can also supply the records to be sorted. When Sort/Merge is ready for a record, it calls the owncode procedure, which then passes a record to Sort/Merge.

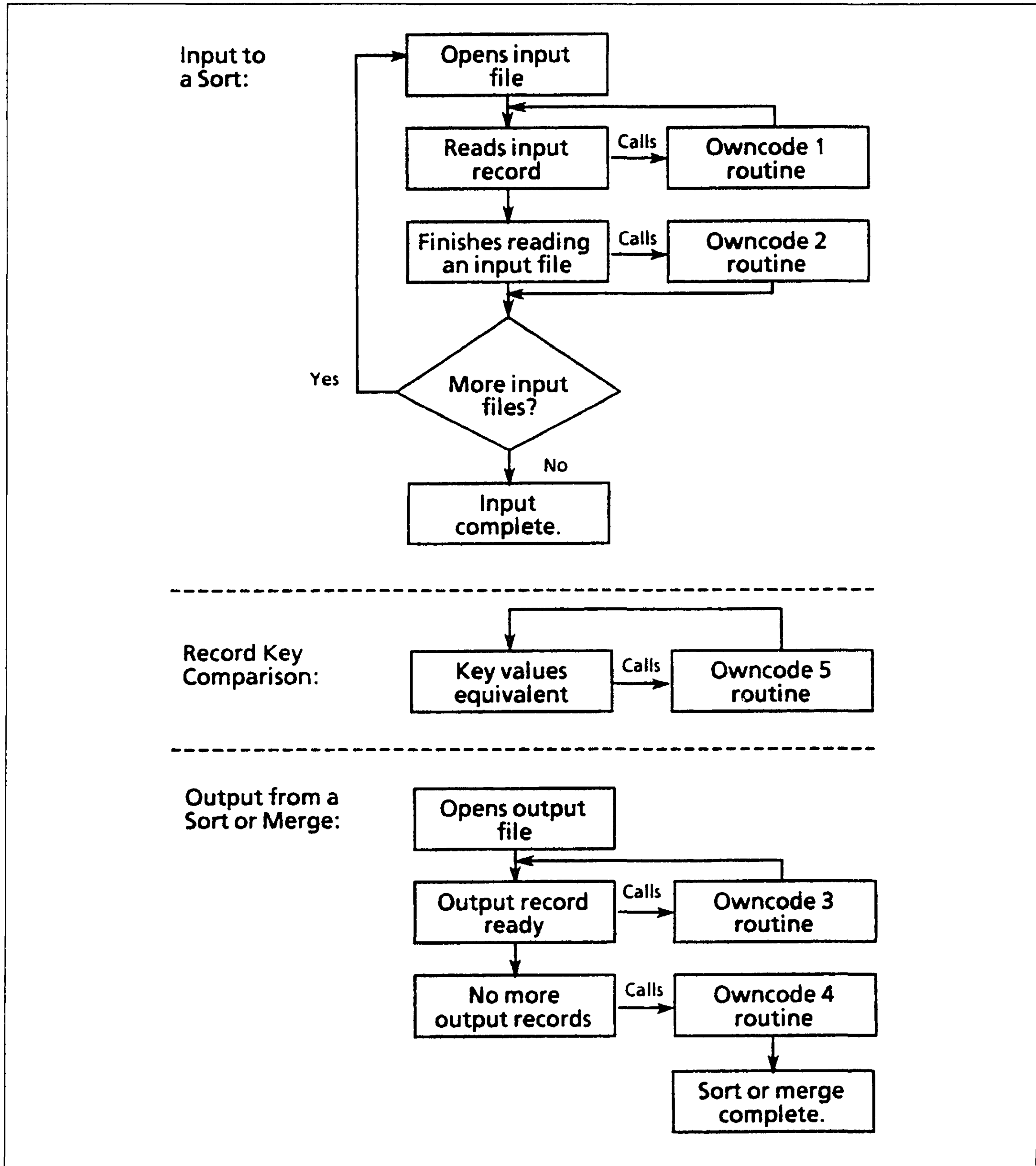


Figure 8-2. When Owncode Procedures are Called

An SM5OWNn call specifies the name of an owncode procedure Sort/Merge is to use; n is an integer from 1 through 5 that tells Sort/Merge at which point in processing the procedure is executed. The SM5OWNn call is described earlier in this chapter.

Owncode procedures 1 and 2 can be called for a sort only; owncode procedures 3, 4, and 5 can be called for a sort or a merge.

SM5OWNn calls are optional. Each SM5OWNn call in the Sort/Merge sequence of calls must specify a different procedure name.

NOTE

When Sort/Merge calls PMP$LOAD to load the owncode procedure, it must pass it a name that uses only uppercase letters. Otherwise, PMP$LOAD cannot find the name in the program library list. Therefore, the user must either specify all owncode procedure names using only uppercase letters or call SM5CC with the TRUE option to convert the names, if necessary.

You can write an owncode procedure using any NOS/VE programming language, including FORTRAN (subprocedure subprograms), COBOL (subprograms compiled with COBOL SP=TRUE option), or CYBIL. The owncode procedure must be compiled and stored as a module in an object library.

Owncode procedures must either be loaded with the main program or be loadable from the program library list. To load an owncode procedure, Sort/Merge calls PMP$LOAD to load the procedure. PMP$LOAD then searches for the specified owncode procedure name in the directories of the object libraries in the program library list.

CYBIL owncode procedures must be declared XDCL procedures.

For Sort/Merge to use an object library containing one or more owncode procedures, the object library file must be in the program library list. To add a file to the program library list before executing the CYBIL program, execute a SET_PROGRAM_ATTRIBUTES command.

For detailed information on creating object libraries, see the NOS/VE Object Code Management Usage manual. The example at the end of this chapter stores an owncode procedure in an object library.

Owncode Procedure Parameters

Sort/Merge communicates with an owncode procedure via the procedure parameter list. Sort/Merge passes record data to the procedure and the procedure returns record data and a code indicating how Sort/Merge is to process the record data.

The following lists the required CYBIL procedure parameter list for owncode 1, owncode 2, owncode 3, and owncode 4 procedures:

```
(VAR return_code: integer;
VAR reca: string(*);
VAR rla: integer);
```

The following lists the required CYBIL procedure parameter list for owncode 5 procedures:

```
(VAR return_code: integer;
VAR reca: string(*);
VAR rla: integer;
VAR recb: string(*);
VAR rlb: integer);
```

The return_code parameter passes an integer code back to Sort/Merge specifying how Sort/Merge is to process the returned records. Sort/Merge always initializes the return_code value to 0 when it calls an owncode procedure. The owncode procedure can leave the return_code value unchanged or change it to one of the valid values for the owncode procedure. (The valid values are listed in the individual owncode procedure description later in this chapter.) If an invalid return_code value is returned, Sort/Merge returns a fatal error.

The subsequent parameters are used to pass one or two records to the owncode procedure. For an owncode 1 through owncode 4 procedure, Sort/Merge passes only one record, the current record being input or output. The record data is passed in the reca variable and the record length in bytes is passed in the rla variable.

When calling an owncode 5 procedure, Sort/Merge passes two records having equal keys. The record data is passed in the reca and recb variables and the corresponding record lengths in the rla and rlb variables.

An owncode procedure can change the record data and record length values passed to it. The procedure must ensure that the record length value returned is correct for the record data returned. However, Sort/Merge does check that the record length returned does not exceed the maximum record length for the sort or merge.

Owncode Procedure Record Length

Sort/Merge checks the length of each record returned to it by an owncode procedure. If a record is too long, Sort/Merge issues an error.

The Sort/Merge specification can explicitly specify the owncode record length. Otherwise, by default, the maximum record length is the largest MAXIMUM_RECORD_LENGTH file attribute value of the input files or output file specified for the sort or merge.

To explicitly specify the owncode record length, you must call SM5OFL or SM5OMRL. If the sort or merge specifies no input or output files, a call to specify the owncode record length is required.

If you call SM5OFL, the length of each record returned by an owncode procedure must exactly match the specified record length value.

If you call SM5OMRL, the length of each record returned by an owncode procedure cannot exceed the specified record length value.

Owncode 1: Processing Input Records

You specify an owncode 1 procedure to process or supply the input records for a sort. An owncode 1 procedure is used only with a sort request; specifying an owncode 1 procedure with a merge request returns a fatal error.

An owncode 1 procedure cannot be used when SMP$FROM_MEMORY is called.

Owncode 1 procedure processing differs depending on whether input files are specified for the sort.

One or More Input Files Specified

If you specify one or more input files for a sort (even if the input file is $NULL), Sort/Merge calls the owncode 1 procedure each time it reads an input record. Sort/Merge passes the input record to the procedure in the reca variable, the record length (in bytes) in the rla variable, and the return_code variable initialized to 0.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 1 procedure. The contents of the reca and rla variables can differ from those originally passed to the procedure.

The following are the valid return_code values and their meanings:

0 Sort/Merge sorts the record passed back in reca and reads the next input record.

1 Sort/Merge does not sort the record in reca and reads the next input record.

2 Sort/Merge sorts the record passed back in reca, but does not read the next input record. Instead, Sort/Merge calls the owncode 1 procedure again so additional records can be added to the sort. The owncode 1 procedure should continue to specify return_code 2 until all records to be inserted at this point have been passed; it should then set the return_code to 0.

3 Sort/Merge does not sort the record passed back in reca, closes the current input file, and calls the owncode 2 procedure if one has been specified. After owncode 2 processing has completed, Sort/Merge opens the next input file, if any, and reads the next input record.

For example, to insert one record after the current input record, the owncode 1 procedure performs the following steps:

1. Checks that the record passed in reca is the record after which the new record is to be inserted.

2. Sets the return_code value to 2 and returns control to Sort/Merge.

3. When called again, it stores the new record in reca, stores the length of the new record in rla, sets the return_code value to 0, and returns control to Sort/Merge.

Input Files Not Specified

If you do not specify any input files for the sort (SM5FROM is not called), Sort/Merge calls the owncode 1 procedure as the source of input records. Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge sorts the record passed back in reca, clears the reca array, sets the rla and return_code variables to 0, and calls the owncode 1 procedure again.

2 Sort/Merge sorts the record passed back in reca, leaves the data in reca and the record length in rla, initializes the return_code to 0, and calls the owncode 1 procedure again.

3 Sort/Merge does not sort the record passed back in reca and calls the owncode 2 procedure if one has been specified; otherwise, terminates the input process.

Owncode 2: Processing Input Files

You specify an owncode 2 procedure to supply input records at the end of each input file. An owncode 2 procedure is used only with a sort request; specifying an owncode 2 procedure with a merge request returns a fatal error.

An owncode 2 procedure cannot be used when SM5FMA is used.

Owncode 2 procedure processing differs depending on whether input files are specified for the sort.

One or More Input Files Specified

If you specify one or more input files for the sort (even if the input file is $NULL), Sort/Merge calls the owncode 2 procedure when it terminates input from an input file. It terminates input when it reads an end-of-partition delimiter or the end-of-information or receives a return_code value of 3 from an owncode 1 procedure.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Owncode 2 processing ends; Sort/Merge opens the next input file, if any, and reads the next input record.

1 Sort/Merge sorts the record passed back in reca, and calls the owncode 2 procedure again.

For example, to insert one record at the end of an input file, the owncode 2 procedure performs the following steps:

1. Stores the record in reca, stores the record length in rla, sets the return_code to 1, and returns control.
2. When called again, leaves the return_code value set to 0, and returns control to Sort/Merge.

Input Files Not Specified

If you do not specify any input files for the sort (SM5FROM is not called), Sort/Merge calls the owncode 2 procedure after the owncode 1 procedure returns a return_code value of 3.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Owncode 2 processing ends, signaling the end of the input records for the sort.

1 Sort/Merge sorts the record passed back in reca, and calls the owncode 2 procedure again.

Owncode 3: Processing Output Records

You specify an owncode 3 procedure to process output records from a sort or merge.

An owncode 3 procedure cannot be used when SM5TMA is called.

Owncode 3 procedure processing differs depending on whether an output file is specified for the sort or merge.

Output File Specified

If you specify an output file for the sort or merge (even if it is $NULL), Sort/Merge calls the owncode 3 procedure each time an output record is ready to be written. Sort/Merge passes the output record to the procedure in the reca variable, the record length in bytes in the rla variable, and the return_code variable initialized to 0.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 3 procedure. The contents of the reca and rla variables can differ from those originally passed to the procedure.

The following are the valid return_code values and their meanings:

0 Sort/Merge writes the record passed back in reca to the output file. It then passes the next output record, if any, to the owncode 3 procedure.

1 Sort/Merge does not write the record passed back in reca to the output file. It then passes the next output record, if any, to the owncode 3 procedure.

2 Sort/Merge writes the record passed back in reca to the output file, leaves the data in reca and the record length in rla, initializes the return_code to 0, and calls the owncode 3 procedure again.

3 Sort/Merge does not write the record passed back in reca. It calls the owncode 4 procedure if one is specified; otherwise, it terminates the sort or merge.

For example, to insert one record after the current output record, the owncode 3 procedure performs the following steps:

1. Checks that the record passed in reca is the record after which the new record is to be inserted.
2. Sets the return_code value to 2 and returns control to Sort/Merge.
3. When called again, stores the new record in reca, stores the length of the new record in rla, sets the return_code value to 0, and returns control to Sort/Merge.

Output File Not Specified

If you do not specify an output file (you do not call SM5TO call for the sort or merge), the owncode 3 procedure performs all output processing. Sort/Merge passes each output record to the owncode 3 procedure, but it does not process any record returned by the procedure. It does not write any output records.

Sort/Merge passes the output record to the procedure in the reca variable, the record length in bytes in the rla variable, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge calls the procedure again, passing the next output record.

1 Sort/Merge calls the procedure again, passing the next output record.

2 Sort/Merge calls the procedure again, passing the same output record.

3 Sort/Merge terminates the output process, even if it has additional output records. It then calls the owncode 4 procedure if one is specified; otherwise, the sort or merge is terminated.

Owncode 4: Processing the Output File

You specify an owncode 4 procedure to write additional output records to the end of the output file. An owncode 4 procedure can be used with a sort or merge.

An owncode 4 procedure cannot be used when SM5TMA is called.

Owncode 4 procedure processing differs depending on whether an output file is specified for the sort or merge.

Output File Specified

If you specify an output file for the sort or merge (even if it is $NULL), Sort/Merge calls the owncode 4 procedure after it has written its last output record to the output file.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge terminates the sort or merge without writing the record passed back in reca.

1 Sort/Merge writes the record passed back in reca and calls the owncode 4 procedure again.

Output File Not Specified

An owncode 4 procedure cannot supply additional output records when no output file has been specified. Still, if you specify an owncode 4 procedure for a sort or merge without an output file, Sort/Merge calls the owncode 4 procedure after the owncode 3 procedure (if any) has terminated output.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge terminates the sort or merge.

1 Sort/Merge terminates the sort or merge.

Owncode 5: Processing Records With Equal Keys

When an owncode 5 procedure is specified, Sort/Merge calls the owncode 5 procedure each time it compares the key values of two records and finds that the values are equivalent. It passes both records to the owncode 5 procedure for processing. An owncode 5 procedure is specified by an SM5OWN5 call.

NOTE

Sort/Merge can interpret character key values as equivalent that are not identical. When the collating sequence used for the key assigns the same collating weight to more than one character, those characters are equivalent key values.

An owncode 5 procedure cannot be used when the SM5SUM, SM5RETA, or SM5OMIT call is used. A sort or merge can use only one method of processing records with equivalent key values.

For a given number (n) of records with equivalent key values, each record is passed to the owncode 5 procedure log n times (assuming that duplicate records are not deleted). The order in which the records are passed is not defined.

NOTE

An owncode 5 procedure can change the record data passed to it, but it must not change the data in the key fields of the record. If it does so, the sort order of the modified key field is undefined.

The following are the valid return_code values for an owncode 5 procedure and the meaning of each:

0 Sort/Merge accepts the first rla bytes of reca as the first record and the first rlb bytes of recb as the second record.

1 Sort/Merge accepts the first rla bytes of reca as the first record and deletes recb from the sort or merge.

2 Sort/Merge accepts the first rlb bytes of recb as the first record and the first rla bytes of reca as the second record.

3 Sort/Merge accepts the first rlb bytes of recb as the first record and deletes reca from the sort or merge.

4 Sort/Merge deletes both records from the sort or merge.

5 Sort/Merge does not read the record data returned by the procedure; it processes the two records in their original order (reca before recb).

6 Sort/Merge does not read the record data returned by the procedure, but it deletes the second record (recb) from the sort or merge.

7 Sort/Merge does not read the record data returned by the procedure, but it reverses the order of the two records (recb before reca).

8 Sort/Merge does not read the record data returned by the procedure, but it deletes the first record (reca) from the sort or merge.

For Better Performance

When the owncode 5 procedure does not change the record data, it should use return_code values 5, 6, 7, or 8 instead of return_code values 0, 1, 2, or 3. Performance is improved because Sort/Merge does not read the returned record data.

Do not use return_code 0 to reverse the order of the two records by exchanging the contents of reca and recb. Performing an exchange sort is both incompatible with and much slower than the Sort/Merge sorting algorithm.

If the owncode 5 procedure sorts the two records using one or more keys in addition to those specified for the sort or merge, the procedure should use return_code values 5 and 7 only. (Return_code values 0 and 2 could also be used, but performance would be slower.)

Using FORTRAN Procedure Calls

This section shows two examples of using Sort/Merge procedure calls in FORTRAN programs. The first example reads a file, selects certain records for processing, sorts them, and writes them to a file. The second example uses an in-memory sort to sort character strings.

Example 1: Sorting the Dean's List

A FORTRAN program DLIST containing the Sort/Merge procedure calls is shown below. File UNIVERSITY_STUDENTS is read, and student records with grade point average of 3.50 or better are written to an intermediate file (INT1). Sort/Merge is called to sort the file on grade point average in descending order (highest grade point average to lowest grade point average).

NOTE

File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $OUTPUT, are assumed to be in the $LOCAL catalog.

```
C
      PROGRAM DLIST
C
C     This program calls Sort/Merge using FORTRAN procedure
C     calls. The purpose of the program is to prepare a
C     list of students with grade point averages of 3.50
C     or better, sort the file on grade point averages in
C     descending order, replace the class code number with
C     the class level, and output the completed report to a
C     new file.
C
C
      INTEGER gpa
      CHARACTER sname*14, major*8, code*1, class*12
      DIMENSION iarray(16)
C
C
      OPEN (1,FILE='university_students')
      REWIND (UNIT=1)
      OPEN (2,FILE='completed_deans_list')
      OPEN (4,FILE='int1')
C
    1 READ (1,100,END=10) sname, major, gpa, code
      IF (gpa .GE. 350) WRITE (4,200) sname, major, gpa, code
      GO TO 1
C
   10 CONTINUE
      CLOSE (UNIT=4,STATUS='KEEP')
C
      IARRAY(1)=15
      CALL SM5SORT (iarray)
      CALL SM5LIST ('$OUTPUT')
      CALL SM5OMRL (80)
      CALL SM5FROM ('int1')
      CALL SM5KEY (33,3,'NUMERIC_NS','D')
      CALL SM5OWN1 ('CCODE')
      CALL SM5TO ('int2')
      CALL SM5END
C
      OPEN (3,FILE='int2')
      REWIND (3)
C
      WRITE (2,400)
   15 READ (3,300,END=20) sname, major, gpa, class
      WRITE (2,500) sname, major, class, gpa
      GO TO 15
C
  100 FORMAT (A14,12X,A8,I3,A1)
  200 FORMAT (A14,5X,A8,5X,I3,5X,A1,39X)
  300 FORMAT (A14,5X,A8,5X,I3,5X,A12,28X)
  400 FORMAT (36X,'DEANS LIST' // 15X, 'STUDENT',
     *        12X,'MAJOR',8X,'CLASS',12X,'GPA',65X /)
  500 FORMAT (15X,A14,5X,A8,5X,A12,5X,I3,59X)
C
   20 STOP
      END
```

The SM5OWN1 call specifies that an owncode 1 procedure named CCODE is to be executed after Sort/Merge reads each record from INT1. Records are passed to the procedure by Sort/Merge. The FORTRAN owncode procedure is shown below.

```
C
C     This is the FORTRAN owncode procedure that is executed
C     after Sort/Merge reads a record.  This procedure
C     replaces the number class code with the class
C     level in words.
C     SUBROUTINE CCODE (retcode,rec,rl)
      INTEGER retcode, rl
      CHARACTER code*1, class*12, rec*(*)
C
      code = rec(41:41)
      IF (code .EQ. '1') THEN
         class = 'SENIOR'
      ELSE IF (code .EQ. '2') THEN
         class = 'JUNIOR'
      ELSE IF (code .EQ. '3') THEN
         class = 'SOPHOMORE'
      ELSE IF (code .EQ. '4') THEN
         class = 'FRESHMAN'
      ELSE IF (code .EQ. '5^) THEN
         class = 'UNCLASSIFIED'
      ELSE
         PRINT *, code
C
      END IF
      rec(41:53) = class
C
C     Set the record length for extra length of class level.
      RL = 53
C
      RETURN
      END
```

The SUBROUTINE statement names the procedure and the parameters passed by Sort/Merge. Parameter RETCODE is the return_code passed as 0, REC is an array containing the record, and RL is the record length in characters. The procedure converts the class code in each record to the class name.

The records are returned to Sort/Merge in array REC. The return_code value is left as 0 because each record in this example is to be sorted. The record received by the owncode procedure is lengthened (RL=53) because the class code is converted into a word and needs more space. Sort/Merge then sorts the record to file INT2. The sorted file is returned to the FORTRAN program to be written out in a formatted report. The content of the intermediate files, INT1 and INT2, is shown below. Figure 8-3 shows the output from the job, which is the completed dean's list report.

```
TERRELL    T H    ENG        386    1
SUGARMAN   B T    SOC        350    1
SMITH      C R    MATH       379    1
SHIELDS    L E    COMPSCI    390    1
DAVIS      D A    ENR        354    1
FRANKLIN   R H    PHIL       370    2
CLARK      D N    ECON       378    2
TIEMON     H R    LNGUIS     376    3
HANSEN     R P    BUS        358    3
SMITH      F R    PHIL       385    3
HORNE      D N    COMPSCI    389    4

SHIELDS    L E    COMPSCI    390    SENIOR
HORNE      D N    COMPSCI    389    FRESHMAN
TERRELL    T H    ENG        386    SENIOR
SMITH      F R    PHIL       385    SOPHOMORE
SMITH      C R    MATH       379    SENIOR
CLARK      D N    ECON       378    JUNIOR
TIEMON     H R    LNGUIS     376    SOPHOMORE
FRANKLIN   R H    PHIL       370    JUNIOR
HANSEN     R P    BUS        358    SOPHOMORE
DAVIS      D A    ENR        354    SENIOR
SUGARMAN   B T    SOC        350    SENIOR
```

```
                        DEANS LIST

STUDENT                 MAJOR          CLASS          GPA

SHIELDS    L E          COMPSCI        SENIOR         390
HORNE      D N          COMPSCI        FRESHMAN       389
TERRELL    T H          ENG            SENIOR         386
SMITH      C R          PHIL           SOPHOMORE      385
SMITH      C R          MATH           SENIOR         379
CLARK      D N          ECON           JUNIOR         378
TIEMON     H R          LNGUIS         SOPHOMORE      376
FRANKLIN   R H          PHIL           JUNIOR         370
HANSEN     R P          BUS            SOPHOMORE      358
DAVIS      D A          ENR            SENIOR         354
SUGARMAN   B T          SOC            SENIOR         350
```

Figure 8-3. Output From the FORTRAN Program

Example 2: An In-Memory Sort

This example sorts character strings that are stored in two arrays. It uses the SM5FMA call (sort from memory area) and the SM5TMA (sort to memory area) to specify the arrays.

```
C
      PROGRAM INMEMORYSORT
C
C     Do an in-memory sort using sort/merge calls SM5FMA and SM5TMA.
C     An array of character strings is filled with font names, and
C     the names are sorted in ASCII order and written to another array
C     of character strings.
C
      INTEGER sortstats(18)       ! array to hold sort statistics
      CHARACTER fontsin(20)*15,   ! memory area, contains unsorted fonts
     +          fontsout(20)*15   ! memory area for sorted fonts
C
      DATA fontsin/'Utopia','Garamond','ITC Galliard','ITC Garamond',
     + 'Stempel Garamond','Garamond 3','Goudy Old Style','Palatino',
     + 'Trump Medieval','Weiss','Americana','Caslon','New Baskerville',
     + 'Century Old St','Janson Text','ITC Clearface','Times*Ten',
     + 'Stone Serif','ITC Tiffany','Bodoni'/
C
C     Specify the number of sort statistics desired.  The statistics will
C     be stored in elements 2 through 18 of the array.
C
      sortstats(1) = 17
C
      CALL SM5SORT(sortstats)
      CALL SM5LIST('$OUTPUT')
      CALL SM5FMA(fontsin, 'FIXED', 15, 20)
      CALL SM5KEY(1,15,'ASCII','A')
      CALL SM5TMA(fontsout, 'FIXED', 15)
      CALL SM5END
C
      write (*, *) sortstats
C
      write (*, '(/,A12)') ' Old order...'
      write (*, '('' '', A16)') (fontsin(i), i=1, 20)
C
      write (*, '(/,A12)') ' New order...'
      write (*, '('' '', A16)') (fontsout(i), i=1, 20)
C
      STOP
      END
```

This example shows how to add the required Sort/Merge library to your program attributes, and compile and execute the program:

```
/set_program_attributes add_library=smf$library
/ftn inmemorysort
/lgo
```

The output of this program is as follows:

```
                 SORT/MERGE STATUS                NOS/VE SORT V1.3 88277
                 1989-08-02  08.37.44,10      PAGE 1

 20 records sorted.

NO DIAGNOSTICS
  17 20 0 0 0 20 0 0 0 0 0 0 20 15 15 15 0 0

 Old order..
  Utopia
  Garamond
  ITC Galliard
  ITC Garamond
  Stempel Garamon
  Garamond 3
  Goudy Old Style
  Palatino
  Trump Medieval
  Weiss
  Americana
  Caslon
  New Baskerville
  Century Old St
  Janson Text
  ITC Clearface
  Times*Ten
  Stone Serif
  ITC Tiffany
  Bodoni

 New order..
  Americana
  Bodoni
  Caslon
  Century Old St
  Garamond
  Garamond 3
  Goudy Old Style
  ITC Clearface
  ITC Galliard
  ITC Garamond
  ITC Tiffany
  Janson Text
  New Baskerville
  Palatino
  Stempel Garamon
  Stone Serif
  Times*Ten
  Trump Medieval
  Utopia
  Weiss
```

Figure 8-4. Output From Program INMEMORYSORT

Creating an Object Library

You must place an owncode procedure into an object library when using command calls. A FORTRAN owncode 3 procedure named OWNCODE is shown below. The procedure OWNCODE will delete the first record in a file. The variable COUNT keeps track of the number of times the owncode procedure is entered.

```
SUBROUTINE OWNCODE (retcode,reca,rla)
INTEGER retcode, rla, count
CHARACTER reca*38
DATA count /0/

count = count +1

IF (count.eq.1) THEN
   retcode = 1
ELSE
   retcode = 0
ENDIF

RETURN
END
```

For detailed information on placing a procedure into a library, see the NOS/VE Object Code Management Usage manual. The commands to place OWNCODE into a library named OWN_LIBRARY are shown below.

```
/ftn i=owncode
/create_object_library
COL/add_module library=$local.lgo
COL/generate_library library=$local.own_library
COL/quit
/display_object_library library=$local.own_library ..
../display_option=entry_point



OWNCODE                           - load module


entry points
~~~~~~~~~~~~


OWNCODE

/set_program_attribute add_library=$local.own_library
```

After executing these commands, the procedure OWNCODE can be called from a FORTRAN program. A FORTRAN program calling OWNCODE is shown below.

```
PROGRAM OWN
    ⋮
Call sm5sort(0)
Call sm5from('univer2')
Call sm5to('results')
Call sm5key(1,10,'ascii','a')
Call sm5own3('OWNCODE')
Call sm5end
    ⋮
STOP
END
```

After the FORTRAN program is executed, the file UNIVERSITY_STUDENTS is sorted, with the first record deleted. The sorted records are written to the file RESULTS as shown below.

```
BILLINGS  C Y 101579111855MUS      2965
BRISCOE   J H 102343121157ENVIRO   2544
CARLSON   M K 102126022355ENGIR    3454
CHARLES   S H 101418032459ANTHRO   2453
CLARK     D N 101400102954ECON     3782
CLARK     D V 101023101956ENG      2083
COCHRAN   G L 100725111857BIO      3011
DAVIES    E D 100812080656JOURN    2031
DAVIS     D A 100972071650ENR      3541
  ⋮
WALLIN    G E 101056041659POLISCI  3151
WARNES    D V 102116060861POLISCI  2814
WILSON    W L 101967010261MATH     3454
WONG      S T 101001012755PSYCH    2152
WOO       R M 101315100159BUS      3223
WOODSTOCK C T 101497030160CHEM     3483
YEH       F L 102005120645Art      2764
YOST      D L 100880111158ENG      2582
ZEITZ     F K 100963111858MATH     2612
ZIMMERS   C A 101075063059MATH     2992
```

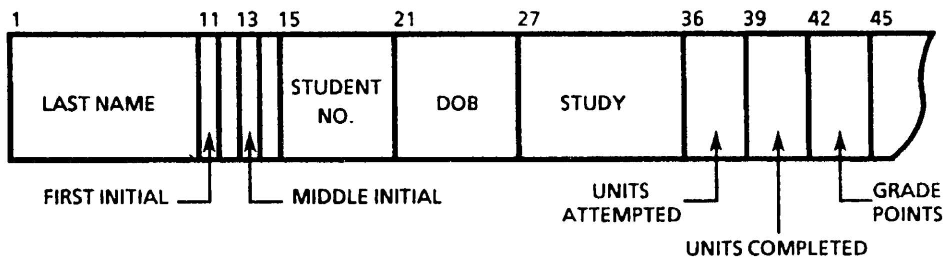
Note that the owncode procedure has deleted the first record in the file.

NOTE

File names referenced without a file path are assumed to be in the working catalog unless the file name is for a standard system file. Standard system files, such as $INPUT or $OUTPUT, are assumed to be in the $LOCAL catalog.

Summing Records

The record layout of a university student file named STUDENTS is shown below.



Each record contains three numeric fields. They are: number of units attempted, number of units completed, and grade points. The file STUDENTS is shown below with multiple records for each student.

```
GREENWOOD M R 102168101961EDU       002002000
IRVING    W R 101750111855ENG       004004016
GREENWOOD M R 102168101961EDU       003003009
IRVING    W R 101750111855ENG       098095375
QUINTERA  L S  90154101253BIO       003000000
ALLEN     M G 102056012561LNGUIS    005000000
ALLEN     M G 102056012561LNGUIS    025020077
ALLEN     M G 102056012561LNGUIS    004004012
```

Records are to be sorted according to the student number. Using the SM5SUM procedure, records with the same student number are combined into one record by adding the numeric fields together. The new record will give the total number of units attempted, total number of units completed, and the total number of grade points.

The procedure to sort and sum the file STUDENTS is as follows:

```
CALL SM5SORT (0)
CALL SM5FROM ('students')
CALL SM5TO ('summed_file')
CALL SM5KEY (15,6,'ascii','a')
CALL SM5SUM (36,3,'numeric_ns',3)
CALL SM5END
```

The input file STUDENTS is named, and the output file SUMMED_FILE will contain the results of the summing. The student number (positions 15 through 20) is specified as the sort key. The SUM procedure specifies that a three-position numeric field of type NUMERIC_NS begins in position 36 in each record. The repetition indicator specifies that three contiguous fields are to be summed. The output from the sort is shown below. Each record ends with nine digits: the first three digits are the total units attempted, the next three are the total units completed, and the final three are the total grade points.

```
QUINTERA  L S  90154101253BIO       003000000
IRVING    W R 101750111855ENG       102099391
ALLEN     M G 102056012561LNGUIS    034024089
GREENWOOD M R 102168101961EDU       005005009
```

The output file contains one record for each student. The numeric fields are the totals of the units attempted, units completed, and grade points.

Defining Your Own Collating Sequence

The file BIRTHDATES, ordered according to the student name, is shown below. The file contains the students' last names, students' first and middle initials, and the students' dates of birth.

```
ALLEN        M G 10-09-61
ANDERSEN     C R 05-01-60
EBERHARD     N I 06 05 58
GREENWOOD    M R 09-12-61
IRVING       W R 01/07/55
KING         M L 11 11 48
QUINTERA     L S 08/12/53
WALLACE      S T 12/09/55
```

You can standardize the separators in the students' birthdate by defining your own collating sequence.

The FORTRAN procedure to define your own collating sequence is as follows:

```
CALL SM5SORT (0)
CALL SM5FROM ('birthdates')
CALL SM5KEY (25,2,'mysequence','a')
CALL SM5KEY (19,3,'mysequence','a')
CALL SM5KEY (22,3,'mysequence','a')
CALL SM5SEQN ('mysequence')
CALL SM5SEQS ('0','1','2')
CALL SM5SEQS ('-',' ','/')
CALL SM5SEQA ('yes')
CALL SM5TO ('dates_sorted')
CALL SM5END
```

The procedure defines a collating sequence named MYSEQUENCE. The first SEQS procedure specifies the digits 0, 1, and 2, all of which will collate equally. The next SEQS parameter specifies one step consisting of hyphens, blanks, and slashes. This defines the hyphen, blank, and slashes as equal values. The SEQA procedure specifies that blanks and slashes are to be output as hyphens. The file is sorted according to the date of birth.

The file DATES_SORTED output from the sort is shown below.

```
KING         M L 11-11-48
QUINTERA     L S 08-12-53
IRVING       W R 01-07-55
WALLACE      S T 12-09-55
EBERHARD     N I 06-05-58
ANDERSEN     C R 05-01-60
GREENWOOD    M R 09-12-61
ALLEN        M G 10-09-61
```

The file BIRTHDATES has been sorted in numeric order according to dates of birth, and the separators in the dates have been changed to hyphens in all records.

Appendixes

Glossary A-1

Related Manuals B-1

ASCII Character Set and Collating Weight Tables C-1

Creating a Collation Table D-1

Differences Between NOS/VE FORTRAN and FORTRAN 5 .... E-1

Glossary A

This section presents a list of definitions of terms used in this manual. Terms are listed in alphabetical order.

A

Access Modes

A file attribute that determines what you can do with the file. Access modes include read, modify, shorten, append, write, execute, and all. Contrast with Share Modes and Open Share Modes.

Advanced Access Methods (AAM)

File manager that processes keyed files.

Alternate Index

Index built in a keyed file for an alternate key. The index associates each alternate key value with a key list of one or more primary-key values.

Alternate Key

Optional key defined in addition to the primary key. An alternate key provides another method of directly accessing records in a keyed file. Unlike the primary key, an alternate key can be defined to allow duplicate values so that more than one record can have the same alternate-key value.

Alternate-Key Definition

Set of attributes that specify alternate-key characteristics. The alternate-key definition is used to build the alternate index for the key.

Ascending Sort Order

Used with the Sort/Merge interface, the order of sorting keys where the record has either a numeric or a non-numeric key and the highest value is written last on the output file. For non-numeric, the first item in the sequence has the lowest value. See Sort Order.

B

Basic Access Methods (BAM)

File manager that processes sequential and byte-addressable files.

Block

In a keyed file, blocks are units of file space linked by pointers.

Byte-Addressable File Organization

File organization in which records are accessed by their byte address in the file.

C

Collated Key

Key type that orders key values according to a user-specified collation table. Contrast with Uncollated Key and Integer Key.

Collating Sequence

Set of values defining the collation weights of the 256 ASCII characters. The collation weights determine the sequence in which characters are ordered and their relative values when compared.

Collation Table

Data structure defining a collating sequence.

Collation Weight

Value assigned to a character that determines the position of that character when ordered using the collating sequence.

D

Data Block

Keyed-file block in which indexed-sequential data records are stored. Contrast with Index Block.

Data Block Padding

Additional space deliberately left in a data block so more data records can be written to the file without a data block split. See Data Block and Data Block Split.

Data Block Split

Process of creating two or three data blocks from an existing data block when a record to be written does not fit into the remaining space of the existing block.

Descending Sort Order

Used with the sort/merge interface, the order of sorting keys where the record has either a numeric or a non-numeric key and the lowest value is written last on the output file. For non-numeric, the first item in the sequence has the highest value. See Sort Order.

Direct Access Input/Output

Method of input/output in which records can be read or written in any order.

Duplicate Key Value

Situation detected when a record to be written to the file has a key value that matches a key value already in the file (or another value for the alternate key in the same record). It can also be detected during application of a new alternate-key definition to a file.

Duplicate Key Value Control

Alternate-key attribute that indicates whether duplicate values are allowed for the key and, if so, how the duplicates are ordered.

E

Embedded Key

Key that is part of the data in each record. (Alternate keys are always embedded.) Contrast with Nonembedded Key.

F

F Record Type

Fixed-length records, as defined by the ANSI standard.

File Information Table (FIT)

Internal table maintained by the FORTRAN keyed-file interface that stores the attributes for an instance of open for a keyed file.

File-Level Parcel

A limited type of parcel that can apply to only one keyed file. Contrast with File-Spanning Parcel.

File Organization

File attribute that determines the record access method for the file. See Sequential File Organization, Byte-Addressable File Organization, and Keyed-File Organization.

File-Spanning Parcel

The type of parcel that can apply to more than one keyed file. Contrast with File-Level Parcel.

FIT

See File Information Table.

H

Hashing Procedure

Procedure used to relate a primary-key value to a home block number in a direct-access file.

Home Block

Unit of space in a direct-access file that can be accessed directly. Initially, a direct-access file is composed of home blocks. Contrast with Overflow Blocks.

I

Index Block

Indexed-sequential file block in which index records are stored. Contrast with Data Block.

Index-Block Padding

Additional space deliberately left in an index block so more records can be written to the file without an index-block split. See Index Block and Index-Block Split.

Index-Block Split

Process of creating two index blocks from an existing index block when a record to be written does not fit into the remaining space of the existing block.

Index Level

Rank in the index block hierarchy in an indexed-sequential file. To find the pointer to a data record, an index block must be searched at each index level.

Index Level Overflow

Condition when a record cannot be written to a file because writing the record would require addition of another index level and the file already has 15 index levels.

Index Record

Record in an index block that associates a key value with a pointer to either a data block or an index block in the next-lower level of the index hierarchy.

Indexed-Sequential File Organization

Keyed-file organization in which records can be read sequentially, ordered by key value, or read randomly by a key value.

Instance of Open

Period of time that a task has a file open. A task may have multiple instances of open. See Task.

Integer Key

Key type that orders key values numerically. The key values can be positive or negative integers. Contrast with Collated Key and Uncollated Key.

J

Job

Set of tasks executed for a user name. See Task.

K

Key

Significant part of a data record.

For Sort/Merge, a key is a part of a record used to determine the position of the record within a sorted sequence of records.

In a keyed file, a key is a part of a record whose value is defined as a means of accessing records. See Primary Key and Alternate Key.

Key List

Sequence of primary-key values associated with an alternate key value in an alternate index. If the alternate key does not allow duplicate values, each key list contains only one value. Otherwise, each key list contains a primary key value for each record that contains the alternate-key value.

Key Type

Kind of data in a key.

For Sort/Merge, a key type is the name of a numeric data format or collating sequence.

For a keyed file, the possible key types are uncollated, collated, and integer.

Keyed-File Organization

File organization that provides for record access by a key value. Currently, the only keyed-file organizations are indexed-sequential and direct-access.

Keyword

Word within a format that must be entered exactly as shown.

L

Lock

Mechanism that makes a primary-key value (or, for a file lock, all primary-key values) inaccessible to other instances of the file.

Log

Entries recording a chronological series of events. The keyed-file interface uses two kinds of logs: parcel logs and update recovery logs. See also Parcel Log and Update Recovery Log.

M

Major Key

Leftmost part of a key. The number of bytes to be used is specified as the major key length. A major key can be used to position or read a keyed file.

Major Sort Key

Used with the Sort/Merge interface, a sort key that is the most important and is specified first.

Merge

Process of combining two or more presorted files.

Minor Sort Key

Used with the Sort/Merge interface, a sort key that is specified after the major sort key on a SORT or MERGE command or in a procedure call. Minor keys are sorted after the major sort key.

N

Nonembedded Key

Primary key that is not part of the record data. Contrast with Embedded Key.

O

Open Share Modes

A file attribute that determines what other instances of open within the same job can do with the file. Open share modes include read, modify, shorten, append, write, execute, and all. Contrast with Access Modes and Share Modes.

Overflow Block

Unit of space in a direct-access file where data is written after a home block fills up. Initially, a direct-access file is composed of home blocks. Contrast with Home Blocks.

Owncode

User-written routine, executed by Sort/Merge, that inserts, substitutes, modifies, or deletes records.

P

Padding

Space deliberately left unused in each block created during the initial open of a keyed file.

Also used to refer to the non-data characters appended to a fixed-length (F) record if the data is shorter than the record length.

Parcel

Series of update operations forming a logical unit. By grouping its update operations into logical units, a program can commit or abort each set of operations as a unit.

Parcel Log

Log in which the system records information from each call to begin, commit, or abort a file-spanning parcel. The program can later fetch the information recorded on the parcel log.

Partition

Unit of data on a sequential or byte-addressable file, delimited by end-of-partition separators or the beginning-of-information or the end-of-information.

Primary Key

Required key in a keyed file. Each primary-key value must be unique in the file. See also Alternate Key.

R

Random Access

Process of reading or writing a record in a file without having to read or write the preceding records; applies only to mass storage files. Contrast with Sequential Access.

Random File Organization

File organization in which records can be accessed by the value of their keys. Random files are processed by direct access READ and WRITE statements, file interface subprograms, and the mass storage subroutines.

Recovery

Actions taken after damage occurs to alleviate the effects of the damage. Keyed-file recovery actions include reloading a backup copy and restoring the copy with an update recovery log . See also Update Recovery Log.

Repeating Groups

Alternate-key attribute indicating that each data record can contain more than one value for the alternate key.

S

Sequential Access

Access mode in which records are processed in the order (physical or logical) in which they occur on a storage device. Contrast with Random Access.

Sequential File Organization

File organization in which records can only be processed in physical order. Records are always read in the order that they were written to the file.

Share Modes

A file attribute that determines what other instances of open can do with the file. Share modes include read, modify, shorten, append, write, execute, and all. Contrast with Access Modes and Open Share Modes.

Sort

Process of arranging records in a specified order.

Sort Key

Used with the Sort/Merge interface, a field of information within each record in a sort or merge input file that is used to determine the order in which records are written to the output file.

Sort Order

Ordering of data according to key fields, either ascending or descending.

Sparse-Key Control

Alternate-key attribute that allows only certain records to be included in the alternate index. Inclusion or exclusion of a record is determined by the character at the sparse-key control position of the record.

T

Task

Instance of execution of a program. Contrast with Job.

U

U Record Type

Records for which the record structure is undefined.

Uncollated Key

Key consisting of 1 to 255 8-bit characters. These keys are sorted by the magnitude of their binary ASCII code values. Contrast with Collated Key and Integer Key.

Update Recovery Log

Log on which each backup or update operation to a keyed file is recorded so that, if the file is damaged, a backup file copy can be reloaded and updated using the information on the log.

V

V Record Type

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

Related Manuals B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the NOS/VE System Usage manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/explain
```

Ordering Printed Manuals

You can order Control Data manuals through Control Data sales offices or through:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Accessing Online Manuals

To access an online manual, log in to NOS/VE and specify the online manual title (listed in table B-1) on the EXPLAIN command. For example, to read the FORTRAN online manual, enter:

```
/help manual=fortran
```

Table B-1. Related Manuals

Manual Title	Publication Number	Online Title
FORTRAN Manuals:		
FORTRAN Version 1 for NOS/VE Language Definition Usage	60485913	
FORTRAN Version 1 for NOS/VE Quick Reference		FORTRAN
FORTRAN for NOS/VE Summary	60485919	
FORTRAN for NOS/VE Tutorial	60485912	FORTRAN_T
FORTRAN for NOS/VE Topics for FORTRAN Programmers Usage	60485916	
FORTRAN Version 2 for NOS/VE Language Definition Usage	60487113	
FORTRAN Version 2 for NOS/VE Quick Reference		VFORTRAN
NOS/VE Manuals:		
NOS/VE Advanced File Management Usage	60486413	AFM
NOS/VE System Usage	60464014	
NOS/VE Commands and Functions	60464018	SCL
NOS/VE Source Code Management Usage	60464313	
NOS/VE Object Code Management Usage	60464413	
Additional References:		
NOS/VE Diagnostic Messages	60464613	MESSAGES
Math Library for NOS/VE Usage	60486513	
Debug for NOS/VE Usage	60488213	
Debug for NOS/VE Quick Reference		DEBUG
Migration From NOS to NOS/VE Tutorial/Usage	60489503	
Programming Environment for NOS/VE Usage		ENVIRONMENT
Professional Programming Environment Usage	60486613	
Professional Programming Environment Quick Reference		PPE

ASCII Character Set and Collating Weight Tables C

Tables C-1 through C-12 give the ASCII character set, the hexadecimal character code for each ASCII character, and the weight tables for the following collating sequences:

- ASCII: FORTRAN default collating sequence
- OSV$ASCII6_FOLDED and OSV$ASCII6_STRICT: NOS FORTRAN 5 default collating sequence.
- OSV$COBOL6_FOLDED and OSV$COBOL6_STRICT: NOS COBOL 5 default collating sequence.
- OSV$DISPLAY63_FOLDED and OSV$DISPLAY63_STRICT: NOS 63-character display code collating sequence.
- OSV$DISPLAY64_FOLDED and OSV$DISPLAY64_STRICT: NOS 64-character display code collating sequence.
- OSV$EBCDIC: Full EBCDIC collating sequence.
- OSV$EBCDIC6_FOLDED and OSV$EBCDIC6_STRICT: EBCDIC 6-bit subset collating sequence supported by NOS COBOL 5 and SORT 5.

The collation table variants FOLDED and STRICT indicate different mapping of the characters not in the 63 or 64 characters of the original NOS collating sequence. A strict mapping maps all characters not in the original 64- or 63-character set to the ordinal for the space character. A folded mapping maps some characters into ordinals of the original characters and the others into the ordinal value for the space character as shown in the listing of the collating sequence.

The following table shows the COLSEQ call parameter values and their corresponding weight table selection:

COLSEQ Call Parameter Value	Selected Collating Weight Table
ASCII	Standard ASCII
ASCII6	OSV$ASCII6_FOLDED
ASCII6S	OSV$ASCII6_STRICT
COBOL6	OSV$COBOL6_FOLDED
COBOL6S	OSV$COBOL6_STRICT
DISPLAY	OSV$DISPLAY64_FOLDED
DISPLAYS	OSV$DISPLAY64_STRICT
DISPLAY63	OSV$DISPLAY63_FOLDED
DISPLAY63S	OSV$DISPLAY63_STRICT
EBCDIC	OSV$EBCDIC
EBCDIC6	OSV$EBCDIC6_FOLDED
EBCDIC6S	OSV$EBCDIC6_STRICT
INSTALL	OSV$COBOL6_FOLDED

Table C-1. ASCII Character Set and Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
0	00	NULL	Null
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	ETX	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tabulation
10	0A	LF	Line feed
11	0B	VT	Vertical tabulation
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledge
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Quotation marks
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Hyphen
46	2E	.	Period
47	2F	/	Slant
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than
61	3D	=	Equal to
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening bracket
92	5C	\	Reverse slant
93	5D	]	Closing bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening brace
124	7C	|	Vertical line
125	7D	}	Closing brace
126	7E	~	Tilde
127	7F	DEL	Delete

ASCII codes 80 through FF hexadecimal (not listed in this table) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-2. OSV$ASCII6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40,60	@,`	Commercial at, grave accent
33	41,61	A,a	Uppercase A, lowercase a
34	42,62	B,b	Uppercase B, lowercase b
35	43,63	C,c	Uppercase C, lowercase c
36	44,64	D,d	Uppercase D, lowercase d
37	45,65	E,e	Uppercase E, lowercase e
38	46,66	F,f	Uppercase F, lowercase f
39	47,67	G,g	Uppercase G, lowercase g

(Continued)

Table C-2. **OSV$ASCII6_FOLDED Collating Sequence** *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48,68	H,h	Uppercase H, lowercase h
41	49,69	I,i	Uppercase I, lowercase i
42	4A,6Ai	J,j	Uppercase J, lowercase j
43	4B,6B	K,k	Uppercase K, lowercase k
44	4C,6C	L,l	Uppercase L, lowercase l
45	4D,6D	M,m	Uppercase M, lowercase m
46	4E,6E	N,n	Uppercase N, lowercase n
47	4F,6F	O,o	Uppercase O, lowercase o
48	50,70	P,p	Uppercase P, lowercase p
49	51,71	Q,q	Uppercase Q, lowercase q
50	52,72	R,r	Uppercase R, lowercase r
51	53,73	S,s	Uppercase S, lowercase s
52	54,74	T,t	Uppercase T, lowercase t
53	55,75	U,u	Uppercase U, lowercase u
54	56,76	V,v	Uppercase V, lowercase v
55	57,77	W,w	Uppercase W, lowercase w
56	58,78	X,x	Uppercase X, lowercase x
57	59,79	Y,y	Uppercase Y, lowercase y
58	5A,7A	Z,z	Uppercase Z, lowercase z
59	5B,7B	[,{	Opening bracket, opening brace
60	5C,7C	\,|	Reverse slant, vertical line
61	5D,7D	],}	Closing bracket, closing brace
62	5E,7E	^,~	Circumflex, tilde
63	5F	_	Underline

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-3. OSV$ASCII6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40	@	Commercial at
33	41	A	Uppercase A
34	42	B	Uppercase B
35	43	C	Uppercase C
36	44	D	Uppercase D
37	45	E	Uppercase E
38	46	F	Uppercase F
39	47	G	Uppercase G

(Continued)

Table C-3. OSV$ASCII6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48	H	Uppercase H
41	49	I	Uppercase I
42	4A	J	Uppercase J
43	4B	K	Uppercase K
44	4C	L	Uppercase L
45	4D	M	Uppercase M
46	4E	N	Uppercase N
47	4F	O	Uppercase O
48	50	P	Uppercase P
49	51	Q	Uppercase Q
50	52	R	Uppercase R
51	53	S	Uppercase S
52	54	T	Uppercase T
53	55	U	Uppercase U
54	56	V	Uppercase V
55	57	W	Uppercase W
56	58	X	Uppercase X
57	59	Y	Uppercase Y
58	5A	Z	Uppercase Z
59	5B	[	Opening bracket
60	5C	\	Reverse slant
61	5D	]	Closing bracket
62	5E	^	Circumflex
63	5F	_	Underline

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-4. OSV$COBOL6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40,60	@,`	Commercial at, grave accent
02	25	%	Percent sign
03	5B,7B	[,{	Opening bracket, opening brace
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C,7C	\,|	Reverse slant, vertical line
11	5E,7E	^,~	Circumflex, tilde
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41,61	A,a	Uppercase A, lowercase a
26	42,62	B,b	Uppercase B, lowercase b
27	43,63	C,c	Uppercase C, lowercase c
28	44,64	D,d	Uppercase D, lowercase d
29	45,65	E,e	Uppercase E, lowercase e
30	46,66	F,f	Uppercase F, lowercase f
31	47,67	G,g	Uppercase G, lowercase g
32	48,68	H,h	Uppercase H, lowercase h
33	49,69	I,i	Uppercase I, lowercase i
34	21	!	Exclamation point
35	4A,6A	J,j	Uppercase J, lowercase j
36	4B,6B	K,k	Uppercase K, lowercase k
37	4C,6C	L,l	Uppercase L, lowercase l
38	4D,6D	M,m	Uppercase M, lowercase m
39	4E,6E	N,n	Uppercase N, lowercase n

(Continued)

Table C-4. OSV$COBOL6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F,6F	O,o	Uppercase O, lowercase o
41	50,70	P,p	Uppercase P, lowercase p
42	51,71	Q,q	Uppercase Q, lowercase q
43	52,72	R,r	Uppercase R, lowercase r
44	5D,7D	],}	Closing bracket, closing brace
45	53,73	S,s	Uppercase S, lowercase s
46	54,74	T,t	Uppercase T, lowercase t
47	55,75	U,u	Uppercase U, lowercase u
48	56,76	V,v	Uppercase V, lowercase v
49	57,77	W,w	Uppercase W, lowercase w
50	58,78	X,x	Uppercase X, lowercase x
51	59,79	Y,y	Uppercase Y, lowercase y
52	5A,7A	Z,z	Uppercase Z, lowercase z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-5. OSV$COBOL6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40	@	Commercial at
02	25	%	Percent sign
03	5B	[	Opening bracket
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C	\	Reverse slant
11	5E	^	Circumflex
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41	A	Uppercase A
26	42	B	Uppercase B
27	43	C	Uppercase C
28	44	D	Uppercase D
29	45	E	Uppercase E
30	46	F	Uppercase F
31	47	G	Uppercase G
32	48	H	Uppercase H
33	49	I	Uppercase I
34	21	!	Exclamation point
35	4A	J	Uppercase J
36	4B	K	Uppercase K
37	4C	L	Uppercase L
38	4D	M	Uppercase M
39	4E	N	Uppercase N

(Continued)

Table C-5. OSV$COBOL6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F	O	Uppercase O
41	50	P	Uppercase P
42	51	Q	Uppercase Q
43	52	R	Uppercase R
44	5D	]	Closing bracket
45	53	S	Uppercase S
46	54	T	Uppercase T
47	55	U	Uppercase U
48	56	V	Uppercase V
49	57	W	Uppercase W
50	58	X	Uppercase X
51	59	Y	Uppercase Y
52	5A	Z	Uppercase Z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant

(Continued)

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B,7B	[,{	Opening bracket, opening brace
50	5D,7D	],}	Closing bracket, closing brace
51	3A	:	Colon
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@,`	Commercial at, grave accent
61	5C,7C	\,|	Reverse slant, vertical line
62	5E,7E	^,~	Circumflex, tilde
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant

(Continued)

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	3A	:	Colon
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B,7B	[,{	Opening bracket, opening brace
50	5D,7D	],}	Closing bracket, closing brace
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@,`	Commercial at, grave accent
61	5C,7C	\,|	Reverse slant, vertical line
62	5E,7E	^,~	Circumflex, tilde
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-10. OSV$EBCDIC Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	9C	---	Unassigned
005	09	HT	Horizontal tabulation
006	86	---	Unassigned
007	7F	DEL	Delete
008	97	---	Unassigned
009	8D	---	Unassigned
010	8E	---	Unassigned
011	0B	VT	Vertical tabulation
012	0C	FF	Form feed
013	0D	CR	Carriage return
014	0E	SO	Shift out
015	0F	SI	Shift in
016	10	DLE	Data link escape
017	11	DC1	Device control 1
018	12	DC2	Device control 2
019	13	DC3	Device control 3
020	9D	---	Unassigned
021	85	---	Unassigned
022	08	BS	Backspace
023	87	---	Unassigned
024	18	CAN	Cancel
025	19	EM	End of medium
026	92	---	Unassigned
027	8F	---	Unassigned
028	1C	FS	File separator
029	1D	GS	Group separator
030	1E	RS	Record separator
031	1F	US	Unit separator
032	80	---	Unassigned
033	81	---	Unassigned
034	82	---	Unassigned
035	83	---	Unassigned
036	84	---	Unassigned
037	0A	LF	Line feed
038	17	ETB	End of transmission block
039	1B	ESC	Escape

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
040	88	---	Unassigned
041	89	---	Unassigned
042	8A	---	Unassigned
043	8B	---	Unassigned
044	8C	---	Unassigned
045	05	ENQ	Enquiry
046	06	ACK	Acknowledge
047	07	BEL	Bell
048	90	---	Unassigned
049	91	---	Unassigned
050	16	SYN	Synchronous idle
051	93	---	Unassigned
052	94	---	Unassigned
053	95	---	Unassigned
054	96	---	Unassigned
055	04	EOT	End of transmission
056	98	---	Unassigned
057	99	---	Unassigned
058	9A	---	Unassigned
059	9B	---	Unassigned
060	14	DC4	Device control 4
061	15	NAK	Negative acknowledge
062	9E	---	Unassigned
063	1A	SUB	Substitute
064	20	SP	Space
065	A0	---	Unassigned
066	A1	---	Unassigned
067	A2	---	Unassigned
068	A3	---	Unassigned
069	A4	---	Unassigned
070	A5	---	Unassigned
071	A6	---	Unassigned
072	A7	---	Unassigned
073	A8	---	Unassigned
074	5B	[	Opening bracket
075	2E	.	Period
076	3C	<	Less than
077	28	(	Opening parenthesis
078	2B	+	Plus
079	21	!	Exclamation point

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
080	26	&	Ampersand
081	A9	---	Unassigned
082	AA	---	Unassigned
083	AB	---	Unassigned
084	AC	---	Unassigned
085	AD	---	Unassigned
086	AE	---	Unassigned
087	AF	---	Unassigned
088	B0	---	Unassigned
089	B1	---	Unassigned
090	5D	]	Closing bracket
091	24	$	Dollar sign
092	2A	*	Asterisk
093	29	)	Closing parenthesis
094	3B	;	Semicolon
095	5E	^	Circumflex
096	2D	-	Hyphen
097	2F	/	Slant
098	B2	---	Unassigned
099	B3	---	Unassigned
100	B4	---	Unassigned
101	B5	---	Unassigned
102	B6	---	Unassigned
103	B7	---	Unassigned
104	B8	---	Unassigned
105	B9	---	Unassigned
106	7C	|	Vertical line
107	2C	,	Comma
108	25	%	Percent sign
109	5F	_	Underline
110	3E	>	Greater than
111	3F	?	Question mark
112	BA	---	Unassigned
113	BB	---	Unassigned
114	BC	---	Unassigned
115	BD	---	Unassigned
116	BE	---	Unassigned
117	BF	---	Unassigned
118	C0	---	Unassigned
119	C1	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
120	C2	---	Unassigned
121	60	`	Grave accent
122	3A	:	Colon
123	23	#	Number sign
124	40	@	Commercial at
125	27	'	Apostrophe
126	3D	=	Equals
127	22	"	Quotation marks
128	C3	---	Unassigned
129	61	a	Lowercase a
130	62	b	Lowercase b
131	63	c	Lowercase c
132	64	d	Lowercase d
133	65	e	Lowercase e
134	66	f	Lowercase f
135	67	g	Lowercase g
136	68	h	Lowercase h
137	69	i	Lowercase i
138	C4	---	Unassigned
139	C5	---	Unassigned
140	C6	---	Unassigned
141	C7	---	Unassigned
142	C8	---	Unassigned
143	C9	---	Unassigned
144	CA	---	Unassigned
145	6A	j	Lowercase j
146	6B	k	Lowercase k
147	6C	l	Lowercase l
148	6D	m	Lowercase m
149	6E	n	Lowercase n
150	6F	o	Lowercase o
151	70	p	Lowercase p
152	71	q	Lowercase q
153	72	r	Lowercase r
154	CB	---	Unassigned
155	CC	---	Unassigned
156	CD	---	Unassigned
157	CE	---	Unassigned
158	CF	---	Unassigned
159	D0	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
160	D1	---	Unassigned
161	7E	---	Unassigned
162	73	s	Lowercase s
163	74	t	Lowercase t
164	75	u	Lowercase u
165	76	v	Lowercase v
166	77	w	Lowercase w
167	78	x	Lowercase x
168	79	y	Lowercase y
169	7A	z	Lowercase z
170	D2	---	Unassigned
171	D3	---	Unassigned
172	D4	---	Unassigned
173	D5	---	Unassigned
174	D6	---	Unassigned
175	D7	---	Unassigned
176	D8	---	Unassigned
177	D9	---	Unassigned
178	DA	---	Unassigned
179	DB	---	Unassigned
180	DC	---	Unassigned
181	DD	---	Unassigned
182	DE	---	Unassigned
183	DF	---	Unassigned
184	E0	---	Unassigned
185	E1	---	Unassigned
186	E2	---	Unassigned
187	E3	---	Unassigned
188	E4	---	Unassigned
189	E5	---	Unassigned
190	E6	---	Unassigned
191	E7	---	Unassigned
192	7B	{	Opening brace
193	41	A	Uppercase A
194	42	B	Uppercase B
195	43	C	Uppercase C
196	44	D	Uppercase D
197	45	E	Uppercase E
198	46	F	Uppercase F
199	47	G	Uppercase G

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
200	48	H	Uppercase H
201	49	I	Uppercase I
202	E8	---	Unassigned
203	E9	---	Unassigned
204	EA	---	Unassigned
205	EB	---	Unassigned
206	EC	---	Unassigned
207	ED	---	Unassigned
208	7D	}	Closing brace
209	4A	J	Uppercase J
210	4B	K	Uppercase K
211	4C	L	Uppercase L
212	4D	M	Uppercase M
213	4E	N	Uppercase N
214	4F	O	Uppercase O
215	50	P	Uppercase P
216	51	Q	Uppercase Q
217	52	R	Uppercase R
218	EE	---	Unassigned
219	EF	---	Unassigned
220	F0	---	Unassigned
221	F1	---	Unassigned
222	F2	---	Unassigned
223	F3	---	Unassigned
224	5C	\	Reverse slant
225	9F	---	Unassigned
226	53	S	Uppercase S
227	54	T	Uppercase T
228	55	U	Uppercase U
229	56	V	Uppercase V
230	57	W	Uppercase W
231	58	X	Uppercase X
232	59	Y	Uppercase Y
233	5A	Z	Uppercase Z
234	F4	---	Unassigned
235	F5	---	Unassigned
236	F6	---	Unassigned
237	F7	---	Unassigned
238	F8	---	Unassigned
239	F9	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
240	30	0	Zero
241	31	1	One
242	32	2	Two
243	33	3	Three
244	34	4	Four
245	35	5	Five
246	36	6	Six
247	37	7	Seven
248	38	8	Eight
249	39	9	Nine
250	FA	---	Unassigned
251	FB	---	Unassigned
252	FC	---	Unassigned
253	FD	---	Unassigned
254	FE	---	Unassigned
255	FF	---	Unassigned

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E,7E	^,~	Circumflex, tilde
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40,60	@,`	Commercial at, grave accent
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B,7B	[,{	Opening bracket, opening brace
26	41,61	A,a	Uppercase A, lowercase a
27	42,62	B,b	Uppercase B, lowercase b
28	43,63	C,c	Uppercase C, lowercase c
29	44,64	D,d	Uppercase D, lowercase d
30	45,65	E,e	Uppercase E, lowercase e
31	46,66	F,f	Uppercase F, lowercase f
32	47,67	G,g	Uppercase G, lowercase g
33	48,68	H,h	Uppercase H, lowercase h
34	49,69	I,i	Uppercase I, lowercase i
35	5D,7D	],}	Closing bracket, closing brace
36	4A,6A	J,j	Uppercase J, lowercase j
37	4B,6B	K,k	Uppercase K, lowercase k
38	4C,6C	L,l	Uppercase L, lowercase l
39	4D,6D	M,m	Uppercase M, lowercase m

(Continued)

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E,6E	N,n	Uppercase N, lowercase n
41	4F,6F	O,o	Uppercase O, lowercase o
42	50,70	P,p	Uppercase P, lowercase p
43	51,71	Q,q	Uppercase Q, lowercase q
44	52,72	R,r	Uppercase R, lowercase r
45	5C,7C	\,|	Reverse slant, vertical line
46	53,73	S,s	Uppercase S, lowercase s
47	54,74	T,t	Uppercase T, lowercase t
48	55,75	U,u	Uppercase U, lowercase u
49	56,76	V,v	Uppercase V, lowercase v
50	57,77	W,w	Uppercase W, lowercase w
51	58,78	X,x	Uppercase X, lowercase x
52	59,79	Y,y	Uppercase Y, lowercase y
53	5A,7A	Z,z	Uppercase Z, lowercase z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E	^	Circumflex
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40	@	Commercial at
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B	[	Opening bracket
26	41	A	Uppercase A
27	42	B	Uppercase B
28	43	C	Uppercase C
29	44	D	Uppercase D
30	45	E	Uppercase E
31	46	F	Uppercase F
32	47	G	Uppercase G
33	48	H	Uppercase H
34	49	I	Uppercase I
35	5D	]	Closing bracket
36	4A	J	Uppercase J
37	4B	K	Uppercase K
38	4C	L	Uppercase L
39	4D	M	Uppercase M

(Continued)

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E	N	Uppercase N
41	4F	O	Uppercase O
42	50	P	Uppercase P
43	51	Q	Uppercase Q
44	52	R	Uppercase R
45	5C	\	Reverse slant
46	53	S	Uppercase S
47	54	T	Uppercase T
48	55	U	Uppercase U
49	56	V	Uppercase V
50	57	W	Uppercase W
51	58	X	Uppercase X
52	59	Y	Uppercase Y
53	5A	Z	Uppercase Z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Creating a Collation Table D

One of the key types supported by the keyed-file interface is collated keys. The order in which collated keys are sorted is determined by a collation table. If you specify this key type, you must supply an explicit collation table; there is no system-supplied default collation table. A fatal error occurs if the KEY_TYPE attribute for a file is collated and the file is opened without a collation table supplied.

You can specify a collation table using the $COLLATE_TABLE_NAME keyword or the $COLLATE_TABLE keyword. You specify the keywords on a CALL STOREF statement before the file is opened for its creation run. NOS/VE supplies eleven predefined collation tables; you can specify a predefined collation table or a collation table that you have created.

Predefined Collation Tables

You specify a predefined collation table by specifying its name on a CALL STOREF statement with the $COLLATE_TABLE_NAME keyword. For example, the following statement specifies OSV$ASCII6_FOLDED as the collation table name in ISFIT.

```
CALL STOREF(ISFIT,'$COLLATE_TABLE_NAME','OSV$ASCII6_FOLDED')
```

A collation table name should be entered using only uppercase letters.

The collating sequences of the predefined collation tables are listed in appendix C, ASCII Character Set and Collating Weight Tables.

Creating Your Own Collation Table

The easiest way to create your own collation table within a FORTRAN program is to use the subprograms described under Collating Sequence Control in chapter 9 of the FORTRAN for NOS/VE Version 1 or Version 2 Language Definition manuals. These subprograms create a collation table from a string of characters.

NOTE

For these subprograms to be effective, you must include a $C COLLATE(USER) directive in your program or specify DEFAULT_COLLATION=USER on the FORTRAN command that compiles the program.

To create a collation table, you assign a character to each position within a 256-character string. The characters assigned must comprise the ASCII character set listed in appendix C, ASCII Character Set and Collating Weight Tables. Nonprintable characters are indirectly assigned to their position in the string using the CHAR function. (Before using the CHAR function, specify ASCII on a COLSEQ call as shown in the collation table example.)

The order in which you assign characters to the string is the order in which you want the characters collated. For example, to collate in reverse order, you would assign the characters in reverse order from the order in which the characters are listed in appendix C, ASCII Character Set and Collating Weight Tables. After assigning 256 characters to the string, you call the CSOWN subprogram described in the FORTRAN for NOS/VE Version 1 or Version 2 Language Definition manuals to define the string as the user-specified collating sequence.

The user-specified collating sequence within a FORTRAN program is referenced by the name FTV$USER_COLLATE_TABLE. Therefore, to assign the collating sequence you defined to the $COLLATE_TABLE_NAME file attribute, you specify FTV$USER_COLLATE_TABLE as the collation table name on the STOREF call. For example, the following statement specifies the $COLLATE_TABLE_NAME value for ISFIT.

```
CALL STOREF (ISFIT,'$COLLATE_TABLE_NAME','FTV$USER_COLLATE_TABLE')
```

NOTE

The name FTV$USER_COLLATE_TABLE must be in uppercase letters because it must match a corresponding internal entry point in the FORTRAN run-time routine that handles collation control.

The CALL STOREF statement must appear before the file is first opened for its creation run. When the CALL OPENM statement opens the file, the value of the $COLLATE_TABLE_NAME attribute becomes permanent. Subsequent jobs that read or update the file cannot change the collation table stored with the file. A CALL STOREF statement that attempts to change the collation table is diagnosed as an error.

Collation Table Example

The program in figure D-1 creates and uses a collation table. Note the placement and use of the C$ COLLATE, CALL STOREF, and CALL TABLE statements.

Output from the program is shown in figure D-2. The first part of the output prints each record and key as records are written to the file. After the records are written to the file, the file is closed, opened, and read sequentially. The second part of the output shows the result of the sequential read. The records are in order according to the collating sequence defined in the collation table.

```
      Program CTABLE

C  ****************************************************
C  *  This program creates an indexed sequential file  *
C  *  (IS_FILE) from a sequential file (DATA_FILE).    *
C  *  Also, this program shows how to set up and use a *
C  *  collation table through a FORTRAN program.       *
C  ****************************************************
C  Issue directive telling the compiler that the collation table
C  is user specified.
C$    Collate (user)
C  Declare variables.
      Integer isfit, reclg, stat
      Common iswsa
      Character * 65 iswsa
C  Call subroutine to create the collation table.
      Call Table
C  Set file attributes before opening IS_FILE.
      Call Fileis (isfit,'$LOCAL_FILE_NAME','IS_FILE',
     +           '$MAXIMUM_RECORD_LENGTH',65,'$RECORD_TYPE','FIXED',
     +           '$KEY_LENGTH',20,'$KEY_TYPE','C',
     +           '$KEY_POSITION',0,'$EMBEDDED_KEY','YES',
     +           '$INDEX_PADDING',10,'$DATA_PADDING',15,
     +           '$ESTIMATED_RECORD_COUNT',30)
C  Store the name of the collate table in $COLLATE_TABLE_NAME attribute.
      Call Storef (isfit,'$COLLATE_TABLE_NAME','FTV$USER_COLLATE_TABLE')
C  Open DATA_FILE and IS_FILE.  Check for error on IS_FILE open.
      Open (2,file='DATA_FILE')
      Call Openm (isfit,'NEW')
      Call IFETCH (isfit,'$ERROR_STATUS', stat)
      If (stat.ne.0) go to 90
C  Read each record from DATA_FILE into the working storage area (iswsa),
C  and then put record into IS_FILE.  After put, check for error.  If no
C  error occurred, print the record.
```

Figure D-1. Creation Program

(Continued)

(Continued)

```
   10 Continue
        Read (2,'(A65)',End=30) iswsa
        Call Put (isfit,iswsa)
        Call IFETCH (isfit,'$ERROR_STATUS',stat)
        If (stat.ne.0) go to 90
        Print '(1X,A65)', iswsa
        Go to 10
C  When all records in DATA_FILE have been read, close IS_FILE and
C  check whether error occurred during CLOSE.

   30 Continue
      Call Closem (isfit)
      Call IFETCH (isfit,'$ERROR_STATUS', stat)
      If (stat.eq.0) Go to 40
      Print 900, stat
      Stop
C  Now read IS_FILE.
   40 Continue
      Call Openm (isfit,'input')
      Call Storef (isfit,'$WORKING_STORAGE_ADDRESS',iswsa)
      Call Storef (isfit,'$WORKING_STORAGE_LENGTH',80)
   50 Continue
        Call Getn (isfit)
        Call IFETCH (isfit,'$ERROR_STATUS', stat)
        If (stat.ne.0) Go to 90
        Call IFETCH (isfit,'$FILE_POSITION', filepos)
        If (filpos.eq.64) Go to 70
        Print '('' Record = '',A65)',iswsa
        Go to 50
C  Close IS_FILE and stop.
   70 Continue
      Call Closem (isfit)
      Call IFETCH (isfit,'$ERROR_STATUS', stat)
      If (stat.ne.0) Print '(1X,I6)', stat
      Stop

C  If error occurs during OPEN or PUT, control transfers to this point
C  in program.  The error number is printed and the file is closed.
   90 Continue
      Print 900, stat
      Call Closem (isfit)
      Stop
  900 Format (1X,I6)
      End
C   The following section contains subroutine TABLE.
      Subroutine Table
```

Figure D-1. Creation Program

(Continued)

(Continued)

```
C  ***********************************************************************
C  * This subroutine sets up the collation table.  The user MUST specify *
C  * a collation table if the key type is collated.                      *
C  ***********************************************************************
C$    collate (user)
      Character user*256
C  The following section puts all ascii characters into a string structure.
C  Symbols and numbers on the far right show the ascii graphics (or their
C  abbreviations) and corresponding decimal representations.  Nonprintable
C  characters have to be indirectly assigned into the string by referencing
C  the CHAR function (which MUST be operating in ASCII mode by COLSEQ).
      Call colseq ('ascii')
      User(1:1) = '$'                                              $   36
      User(2:2) = 'R'                                              R   65
      User(3:3) = char(9)                                         HT    9
      User(4:4) = '<'                                              <   60
      User(5:5) = 'F'                                              F   82
      User(6:6) = 'd'                                              d  100
      User(7:7) = char(127)                                       RO  127
      User(8:8) = '+'                                              +   43
      User(9:9) = '%'                                              %   37
      User(10:10) = 'x'                                            x  129
      User(11:11) = char(13)                                      CR   13
      User(12:12) = '3'                                            3   51
      User(13:13) = '#'                                            #   35
      User(14:14) = char(26)                                     CUP   26
      User(15:15) = 'V'                                            V   86
      User(16:16) = 'm'                                            m  109
      User(17:17) = '0'                                            0   48
      User(18:18) = char(18)                                     DC2   18
      User(19:19) = ':'                                            :   58
      User(20:20) = '^'                                            ^   94
      User(21:21) = 'r'                                            r  114
      User(22:22) = char(21)                                    SKIP   21
      User(23:23) = '*'                                            *   42
      User(24:24) = 'K'                                            K   75
      User(25:25) = '{'                                            {  123
      User(26:26) = 'c'                                            c   99
      User(27:27) = '6'                                            6   54
      User(28:28) = 'w'                                            w  119
      User(29:29) = 'P'                                            P   80
      User(30:30) = char(16)                                     DLE   16
      User(31:31) = '_'                                            _   95
      User(32:32) = 'A'                                            A   70
      User(33:33) = char(3)                                      ETX    3
      User(34:34) = 'h'                                            h  104
      User(35:35) = 'H'                                            H   72
```

Figure D-1. Creation Program

(Continued)

(Continued)

```
      User(36:36) = 'D'                                          D   68
      User(37:37) = char(2)                                    STX    2
      User(38:38) = 'M'                                          M   77
      User(39:39) = char(29)                                    GS   29
      User(40:40) = ','                                          ,   44
      User(41:41) = 'a'                                          a   97
      User(42:42) = '~'                                          ~  126
      User(43:43) = 'k'                                          k  107
      User(44:44) = '1'                                          1   49
      User(45:45) = ';'                                          ;   59
      User(46:46) = char(23)                                   ETB   23
      User(47:47) = ' '                                              92
      User(48:48) = 'u'                                          u  117
      User(49:49) = '5'                                          5   53
      User(50:50) = 'B'                                          B   66
      User(51:51) = 'f'                                          f  102
      User(52:52) = char(5)                                    ENQ    5
      User(53:53) = '('                                          (   40
      User(54:54) = '7'                                          7   55
      User(55:55) = '&'                                          &   38
      User(56:56) = 'T'                                          T   84
      User(57:57) = 'b'                                          b   98
      User(58:58) = 'Z'                                          Z   90
      User(59:59) = 'o'                                          o  111
      User(60:60) = ' '                                              32
      User(61:61) = char(27)                                   ESC   27
      User(62:62) = char(7)                                    BEL    7
      User(63:63) = ']'                                          ]   93
      User(64:64) = 'X'                                          X   88
      User(65:65) = '2'                                          2   50
      User(66:66) = char(31)                                    US   31
      User(67:67) = 'N'                                          N   78
      User(68:68) = '`'                                          `   96
      User(69:69) = 'q'                                          q  113
      User(70:70) = char(11)                                    VT   11
      User(71:71) = ')'                                          )   41
      User(72:72) = 'J'                                          J   74
      User(73:73) = '}'                                          }  125
      User(74:74) = char(19)                                   DC3   19
      User(75:75) = 'z'                                          z  131
      User(76:76) = 's'                                          s  115
      User(77:77) = 'Q'                                          Q   81
      User(78:78) = '?'                                          ?   63
      User(79:79) = '9'                                          9   57
      User(80:80) = char(12)                                    FF   12
      User(81:81) = 'e'                                          e  101
      User(82:82) = 'G'                                          G   71
      User(83:83) = '='                                          =   61
```

Figure D-1. Creation Program

(Continued)

(Continued)

```
User(84:84) = char(6)                                    ACK    6
User(85:85) = 'U'                                          U   85
User(86:86) = ''''                                         '   39
User(87:87) = '@'                                          @   64
User(88:88) = 'E'                                          E   69
User(89:89) = '4'                                          4   52
User(90:90) = '"'                                          "   34
User(91:91) = char(30)                                    RS   30
User(92:92) = char(4)                                    EOT    4
User(93:93) = 'g'                                          g  103
User(94:94) = '/'                                          /   47
User(95:95) = char(0)                                    NUL    0
User(96:96) = '-'                                          -   45
User(97:97) = 'I'                                          I   73
User(98:98) = '['                                          [   91
User(99:99) = '|'                                          |  124
User(100:100) = char(28)                                  FS   28
User(101:101) = 'j'                                        j  106
User(102:102) = 'v'                                        v  118
User(103:103) = char(1)                                  SOH    1
User(104:104) = '.'                                        .   46
User(105:105) = 'y'                                        y  130
User(106:106) = char(8)                                   BS    8
User(107:107) = '>'                                        >   62
User(108:108) = 'W'                                        W   67
User(109:109) = 'i'                                        i  105
User(110:110) = 'S'                                        S   83
User(111:111) = '!'                                        !   33
User(112:112) = char(24)                                 CLR   24
User(113:113) = 'l'                                        l  108
User(114:114) = 'L'                                        L   76
User(115:115) = 't'                                        t  116
User(116:116) = char(22)                                LCLR   22
User(117:117) = 'n'                                        n  110
User(118:118) = 'O'                                        O   79
User(119:119) = char(17)                                 DC1   17
User(120:120) = char(25)                                RSET   25
User(121:121) = 'C'                                        C   87
User(122:122) = char(15)                                  SI   15
User(123:123) = 'Y'                                        Y   89
User(124:124) = char(10)                                  LF   10
User(125:125) = 'p'                                        p  112
User(126:126) = char(14)                                  SO   14
User(127:127) = char(20)                                 DC4   20
User(128:128) = '8'                                        8   56
```

Figure D-1. Creation Program

(Continued)

(Continued)

```
C     Complete the table using a loop that assigns the leftover characters
C     in reverse order filling the remaining 128 positions.

      Do 5 I = 129,256
    5 User (I:I) = char (256-I+128)
      Call Csown (user)
      Return
      End
```

Figure D-1. Creation Program

```
Algeria                19709000    919591 Algiers      Africa
Australia              14796000   2967895 Canberra     Australia
Austria                74760000     32374 Vienna       Europe
Belguim                 9875000     11781 Brussels     Europe
Canada                 24336000   3851791 Ottawa       NAmerica
China                1053788000   3705390 Beijing      Asia
Denmark                 5157000     16629 Copenhagen   Europe
England                55717000     94226 London       Europe
France                 53844000    211207 Paris        Europe
India                 700734000   1269340 New Delhi    Asia
Ireland                 3349000     27136 Dublin       Europe
Italy                  57513000    116303 Rome         Europe
Ivory Coast             8513000    124503 Abidjan      Africa
Japan                  11878300    143750 Tokyo        Asia
Mexico                 70143000    761601 Mexico City  SAmerica
Spain                  38686000    194897 Madrid       Europe
Sweden                  8335000    173731 Stockholm    Europe
Switzerland            63000000     15941 Bern         Europe
Tanzania               18744000    364898 Zanzibar     Africa
Turkey                 47284000    301381 Ankara       Asia
USSR                  269302000   8649498 Moscow       Asia
United States         225195000   3615105 Washington   NAmerica
Venezuela              15771000    352143 Caracas      SAmerica
West Germany           60948000     95976 Bonn         Europe
-- File IS_FILE : 0 DELETE_KEYs done since last open.
-- File IS_FILE : 0 GET_KEYs done since last open.
-- File IS_FILE : 0 GET_NEXT_KEYs done since last open.
-- File IS_FILE : 24 PUT_KEYs (and PUTREPs->put) since last open.
-- File IS_FILE : 0 PUTREPs done since last open.
-- File IS_FILE : 0 REPLACE_KEYs (and PUTREPs->replace) since last open.
Record = France             53844000    211207 Paris        Europe
Record = Venezuela          15771000    352143 Caracas      SAmerica
Record = Australia          14796000   2967895 Canberra     Australia
```

Figure D-2. Creation Program Output

(Continued)

(Continued)

```
Record = Austria                 74760000     32374 Vienna        Europe
Record = Algeria                 19709000    919591 Algiers       Africa
Record = Denmark                  5157000     16629 Copenhagen    Europe
Record = Mexico                  70143000    761601 Mexico City   SAmerica
Record = Belguim                  9875000     11781 Brussels      Europe
Record = Tanzania                18744000    364898 Zanzibar      Africa
Record = Turkey                  47284000    301381 Ankara        Asia
Record = Japan                   11878300    143750 Tokyo         Asia
Record = USSR                   269302000   8649498 Moscow        Asia
Record = United States          225195000   3615105 Washington    NAmerica
Record = England                 55717000     94226 London        Europe
Record = Ireland                  3349000     27136 Dublin        Europe
Record = Ivory Coast              8513000    124503 Abidjan       Africa
Record = Italy                   57513000    116303 Rome          Europe
Record = India                  700734000   1269340 New Delhi     Asia
Record = West Germany            60948000     95976 Bonn          Europe
Record = Sweden                   8335000    173731 Stockholm     Europe
Record = Switzerland             63000000     15941 Bern          Europe
Record = Spain                   38686000    194897 Madrid        Europe
Record = China                 1053788000   3705390 Beijing       Asia
Record = Canada                  24336000   3851791 Ottawa        NAmerica
--  File IS_FILE : AMP$GET_NEXT_KEY has reached a file boundary : EOI.
--  File IS_FILE :  0 DELETE_KEYs done since last open.
--  File IS_FILE :  0 GET_KEYs done since last open.
--  File IS_FILE :  24 GET_NEXT_KEYs done since last open.
--  File IS_FILE :  0 PUT_KEYs (and PUTREPs->put) since last open.
--  File IS_FILE :  0 PUTREPs done since last open.
--  File IS_FILE :  0 REPLACE_KEYs (and PUTREPs->replace) since last open.
```

Figure D-2. Creation Program Output

Creating a Collation Weight Table

It is also possible to create a collation table to be specified by address using the DCT keyword. The collation table must be in the form of a collation weight table. (The CSOWN subprogram generates a collation weight table from a character string.)

A collation weight table is 256 contiguous bytes (32 words) with each byte containing an integer value. The 256 bytes within the table correspond to the 256 character codes in the ASCII character set. The collation weight, or ordinal, for each character is the value stored in the byte corresponding to the character within the table.

Figure D-3 illustrates the collation weight table for the ASCII collation sequence with the weights in hexadecimal. Weights are assigned in ascending order just as the characters are ordered in the set. The character codes from 80 through FF hexadecimal do not have graphic characters associated with them. However, each character code is assigned a collating weight within the table.

As illustrated, the weights for the uppercase letters are in bytes 41 through 5A hexadecimal of the string and the weights for the lowercase letters are in bytes 61 through 7A.

Suppose you want the lowercase letters to be collated the same as the uppercase letters (case insensitive). You would then assign the collating weight of each uppercase letter to the corresponding lowercase letter. The following is a listing of words 12 through 15 of the collation weight table showing the changed values for the lowercase letters.

	a	b	c	d	e	f	g
60	41	42	43	44	45	46	47

h	i	j	k	l	m	n	o
48	49	4A	4B	4C	4D	4E	4F

p	q	r	s	t	u	v	w
50	51	52	53	54	55	56	57

x	y	z	{	|	}	~	DEL
58	59	5A	7B	7C	7D	7E	7F

To create a collation table, you declare a 32-word integer array and then assign a hexadecimal constant to each word in the array. For example, the following statement declares an array named TABLE with bounds 0 and 31.

```
INTEGER TABLE (0:31)
```

You then assign a hexadecimal constant to each of the 32 words in the array. For example, when creating a case insensitive collation table, you would assign the following hexadecimal constants to words 12 through 15 of the array.

```
TABLE(12) = Z"6041424344454647"

TABLE(13) = Z"48494A4B4C4D4E4F"

TABLE(14) = Z"5051525354555657"

TABLE(15) = Z"58595A7B7C7D7E7F"
```

<table>
<tr><th>Word</th><th></th><th></th><th></th><th></th><th></th><th></th><th></th><th></th></tr>
<tr><td></td><td>NUL</td><td>SOH</td><td>STX</td><td>ETX</td><td>EOT</td><td>ENQ</td><td>ACK</td><td>BEL</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td></td><td>BS</td><td>HT</td><td>LF</td><td>VT</td><td>FF</td><td>CR</td><td>SO</td><td>SI</td></tr>
<tr><td>1</td><td>8</td><td>9</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr>
<tr><td></td><td>DLE</td><td>DC1</td><td>DC2</td><td>DC3</td><td>DC4</td><td>NAK</td><td>SYN</td><td>ETB</td></tr>
<tr><td>2</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><td></td><td>CAN</td><td>EM</td><td>SUB</td><td>ESC</td><td>FS</td><td>GS</td><td>RS</td><td>US</td></tr>
<tr><td>3</td><td>18</td><td>19</td><td>1A</td><td>1B</td><td>1C</td><td>1D</td><td>1E</td><td>1F</td></tr>
<tr><td></td><td>SP</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td></tr>
<tr><td>4</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td></tr>
<tr><td></td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>5</td><td>28</td><td>29</td><td>2A</td><td>2B</td><td>2C</td><td>2D</td><td>2E</td><td>2F</td></tr>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>6</td><td>30</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td></tr>
<tr><td></td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>7</td><td>38</td><td>39</td><td>3A</td><td>3B</td><td>3C</td><td>3D</td><td>3E</td><td>3F</td></tr>
<tr><td></td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr>
<tr><td>8</td><td>40</td><td>41</td><td>42</td><td>43</td><td>44</td><td>45</td><td>46</td><td>47</td></tr>
<tr><td></td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>9</td><td>48</td><td>49</td><td>4A</td><td>4B</td><td>4C</td><td>4D</td><td>4E</td><td>4F</td></tr>
<tr><td></td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td></tr>
<tr><td>10</td><td>50</td><td>51</td><td>52</td><td>53</td><td>54</td><td>55</td><td>56</td><td>57</td></tr>
<tr><td></td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>^</td><td>_</td></tr>
<tr><td>11</td><td>58</td><td>59</td><td>5A</td><td>5B</td><td>5C</td><td>5D</td><td>5E</td><td>5F</td></tr>
<tr><td></td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td>12</td><td>60</td><td>61</td><td>62</td><td>63</td><td>64</td><td>65</td><td>66</td><td>67</td></tr>
<tr><td></td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>13</td><td>68</td><td>69</td><td>6A</td><td>6B</td><td>6C</td><td>6D</td><td>6E</td><td>6F</td></tr>
<tr><td></td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td></tr>
<tr><td>14</td><td>70</td><td>71</td><td>72</td><td>73</td><td>74</td><td>75</td><td>76</td><td>77</td></tr>
<tr><td></td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>~</td><td>DEL</td></tr>
<tr><td>15</td><td>78</td><td>79</td><td>7A</td><td>7B</td><td>7C</td><td>7D</td><td>7E</td><td>7F</td></tr>
</table>

Figure D-3. Collation Weight Table

(Continued)

(Continued)

Word								
16	80	81	82	83	84	85	86	87
17	88	89	8A	8B	8C	8D	8E	8F
18	90	91	92	93	94	95	96	97
19	98	99	9A	9B	9C	9D	9E	9F
20	A0	A1	A2	A3	A4	A5	A6	A7
21	A8	A9	AA	AB	AC	AD	AE	AF
22	B0	B1	B2	B3	B4	B5	B6	B7
23	B8	B9	BA	BB	BC	BD	BE	BF
24	C0	C1	C2	C3	C4	C5	C6	C7
25	C8	C9	CA	CB	CC	CD	CE	CF
26	D0	D1	D2	D3	D4	D5	D6	D7
27	D8	D9	DA	DB	DC	DD	DE	DF
28	E0	E1	E2	E3	E4	E5	E6	E7
29	E8	E9	EA	EB	EC	ED	EE	EF
30	F0	F1	F2	F3	F4	F5	F6	F7
31	F8	F9	FA	FB	FC	FD	FE	FF

Figure D-3. Collation Weight Table

Differences Between NOS/VE FORTRAN and FORTRAN 5 — E

This appendix presents the differences between FORTRAN 5 and the first released version of NOS/VE FORTRAN, and is intended as an aid to converting programs from FORTRAN 5 to NOS/VE FORTRAN.

NOS/VE FORTRAN is designed to be compatible with FORTRAN 5. However, the new operating system and hardware have resulted in several areas of incompatibility. Other incompatibilities are the result of FORTRAN 5 features which are not currently supported under NOS/VE FORTRAN but for which future support is anticipated.

In some cases, language incompatibilities may necessitate program modification; in other cases, statements using incompatible features can remain in the program but will not be processed.

Coding that depends on the internal representation of data (floating-point layout, number of characters per word, and so forth) should be checked. Because of differences in word size and internal representations, these uses nearly always require modification.

Data manipulations based on the binary representation of the data should be checked. FORTRAN 5 programs that manipulate characters as octal display-coded values or as 6-bit binary digits must be modified before being compiled and executed under NOS/VE FORTRAN.

File structure and naming conventions differ significantly under NOS/VE, and default file positioning has changed. All usages that depend on any of these properties should be checked.

CYBER Record Manager (AAM) Subprograms

The capabilities provided by CYBER Record Manager (CRM) are provided by the file interface routines under NOS/VE. As with CRM, all FORTRAN I/O is performed through the file interface, and a set of FORTRAN subprogram calls provides direct communication with the file interface.

Currently, NOS/VE supports only sequential, indexed-sequential, direct access, and byte-addressable file organizations. Only indexed-sequential and direct access files can be accessed by direct FORTRAN calls. Actual-key file organization is not supported. The Basic Access Methods word addressable organization has been replaced by the new byte-addressable organization.

You should check all uses of CRM Advanced Access Methods (AAM) subprogram calls in your FORTRAN programs. The NOS/VE keyed file interface calls offer only a subset of the features offered by the CRM AAM calls. The following paragraphs describe the feature differences.

File Organization

Currently, the only file organizations available via the keyed file interface calls are indexed-sequential and direct access.

Record Type

The record types available are fixed-length (F) and variable-length (U or V). NOS/VE does not support the AAM Version 2 record types D, R, S, T, and Z.

File Information Table

User programs do not need to reserve 35 words for the file information table. All that is needed is room for a one-word pointer. If the program does reserve 35 words, only the first word will be used.

Values can be stored or fetched from the file information table in standard ways, i.e., CALL FILEIS, CALL FILEDA, CALL STOREF, CALL IFETCH, and IFETCH. Values in the file information table can only be modified through the file processing calls because the file information table is an internal table which cannot be accessed directly by a program.

Any attempt to read from the table without using IFETCH returns an undefined value. If a value is stored in an unconventional manner, the value cannot be returned.

Keywords must be enclosed in apostrophes; for example, 'WSA'. The boolean form L"WSA", used in FORTRAN 5 programs, does not work.

The following CYBER Record Manager file information fields do not have equivalents in the file interface to FORTRAN:

BAL	BBH	BFF	BFS	BS	BT
BZF	B8F	CDT	CL	CM	CNF
CP	CPA	C1	DCA	DFLG	DKI
EFC	EO	EOFWA	EXD	FPB	FWB
HB	HL	HRL	IBL	IRS	KNE
KR	LA	LAC	LBL	LCR	LGX
LL	LNG	LOP5	LP	LT	LVL
LX	MFN	MNB	MUL	NDX	NOFCP
ORG	OVF	PC	PEF	PKA	PM
PNO	POS	PTL	RC	RDR	RMK
SB	SBF	SDS	SES	SOL	SPR
TL	TRC	ULP	VF	VNO	WA
WPN	XBS	XN			

Field FL, although not applicable to the file interface to FORTRAN, will be recognized as a synonym of field MRL.

Other keywords from Advanced Access Methods Version 2 and their meanings for the file interface to FORTRAN:

DX Data exit. Although NOS/VE does not support data exit, the FORTRAN keyed file interface saves the subroutine address and calls the subroutine when the appropriate file position (BOI or EOI) is returned from an access operation.

OC Open/closed flag. Although NOS/VE system requests tell whether the file is opened or closed, the file information table will also contain this information so you can read it by a IFETCH operation.

FNF Fatal error flag. To allow you to read the information with a fetch request, the FORTRAN interface maintains this information in the file information table.

ON Old/new flag. The file information table maintains a value of ON which can be set to OLD (default) or NEW by a FILEIS or FILEDA call. When a CALL OPENM statement is issued, the FORTRAN interface first finds out from the system whether the file already exists. If the answer to this question conflicts with the setting of ON, a fatal error occurs.

KP Key position. This keyword, although it has no meaning in AAM NOS/VE, is accepted by the FORTRAN interface as a keyword in the CALL FILEIS or CALL FILEDA statement or as a parameter in the CALL STARTM, CALL STOREF, and CALL GET statements. KP is added to KA to determine the position of the key.

RKW Relative key word. If RKW is specified in a CALL FILEIS or CALL FILEDA statement, the keyed-file interface multiplies the value by 10 and adds it to RKP. This may be a problem because NOS/VE has a word size of 8 bytes and not 10 bytes (NOS and NOS/BE). Users should visually inspect the program to ensure that the correct value is specified.

Reserving Space for WSA

Check whether your FORTRAN 5 program uses an INTEGER or REAL array for WSA. Because NOS and NOS/BE use a 10-byte word and NOS/VE uses an 8-byte word, the number of characters that can be stored in an INTEGER or REAL array differs.

For example, in NOS/VE FORTRAN, coding a statement like RECORD (8) reserves only 64 characters of space (as opposed to 80 characters in FORTRAN 5), and the first time a record is read into the area, the record overwrites the next item in memory.

To write a FORTRAN program in which the same number of characters can be stored in the WSA when the program is executed by NOS, NOS/BE, or NOS/VE, declare the WSA using the CHARACTER data type.

Optimization

FORTRAN optimization (OL=HIGH) can cause unpredictable results when WSA, KA or PKA are not in common. If OL=HIGH is to be used, WSA, PKA and KA should be declared as COMMON.

Embedded Keys

The default for EMK in Advanced Access Methods Version 2 was NO (nonembedded keys). The default for the NOS/VE keyed file interface is YES (embedded keys).

CALL GETNR Statement

For purposes of compatibility, CALL GETNR statement is allowed. CALL GETNR is treated as a CALL GETN.

CALL SEEKF Statement

The SEEK function does not exist in the file interface to FORTRAN. If a CALL SEEKF is encountered, the FORTRAN interface copies parameters to the file information table, sets the FILE_POSITION field to end-of-information (EOR), and returns control to the program.

Default Collating Sequence

The default collating sequence established when the DEFAULT_COLLATION parameter is omitted from the FORTRAN command has been changed from USER to FIXED. Under NOS/VE FORTRAN, the USER and FIXED collating sequences are defined as the 'ASCII' and 'DISPLAY' collating sequences, respectively. Under FORTRAN 5, USER and FIXED are defined as 'DISPLAY' and 'ASCII6', respectively.

See also Other Collating Sequence Differences in this section.

Other Collating Sequence Differences

NOS/VE FORTRAN uses standard system-defined collating sequences for the NOS-compatible 'ASCII6' and 'COBOL6' collating sequences. The 'STANDARD' sequence of FORTRAN 5 has been eliminated, and an 'INSTALL' sequence, equivalent to 'COBOL6', has been added.

Sort/Merge

NOS/VE Sort/Merge is compatible only with Sort/Merge Version 5; it does not attempt compatibility with any other Sort/Merge version. NOS/VE Sort/Merge can only access NOS/VE disk files.

The File Management Utility (FMU) can convert NOS files into equivalent NOS/VE files. This utility converts the differences in byte size, collating sequence, record type, and block type. See the NOS/VE Advanced File Management Usage manual for more details.

The following paragraphs list the major differences between NOS/VE Sort/Merge and NOS Sort/Merge Version 5.

Byte Size

Under NOS/VE Sort/Merge, the byte size is equal to 8-bits rather than 6-bits which is the case under NOS Sort/Merge 5.

Character Codes

Character data is internally represented in 8-bit ASCII character codes under NOS/VE Sort/Merge rather than 6-bit display codes which is the case under NOS Sort/Merge 5.

Character Sets

NOS/VE Sort/Merge supports only the 256-character ASCII character set. NOS/VE Sort/Merge does not support the 63- and 64-character sets.

Collating Sequences

There are now six predefined collating sequences under NOS/VE Sort/Merge: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. ASCII is assumed if a sequence is not specified.

Under NOS/VE a user-defined collating sequence has 256 positions. (NOS/VE can use the SEQR procedure to fill the rest).

Direct Processing

NOS/VE Sort/Merge does not support direct processing (all records are read and written through the access method). NOS Sort/Merge 5 reads and writes directly (instead of through CYBER Record Manager) if so specified by the SM5FAST procedure.

Error File

The default error file is $ERRORS under NOS/VE Sort/Merge.

Error Messages

NOS/VE Sort/Merge error numbers and message text follow NOS/VE error message conventions.

The NOS/VE Sort/Merge error messages are listed in the NOS/VE Diagnostic Messages manual.

Estimated Number of Records

For NOS/VE Sort/Merge, the value can be specified on the SM5ENR procedure call.

Exception File Processing

NOS/VE Sort/Merge performs exception file processing if an exception file is specified for the sort or merge.

File Attributes

The NOS default file attributes are valid for a sort or merge.

The NOS/VE default value for the minimum record length attribute could cause a fatal error if no key field was specified for the sort or merge.

File Manipulation

Files are not rewound by NOS/VE Sort/Merge. The open position of a NOS/VE file is determined by the value of its open_position attribute.

Interactive Prompting

Interactive prompting is not currently implemented on NOS/VE Sort/Merge.

Listing File

NOS/VE Sort/Merge provides the SM5LIST procedure to specify the listing file. The default listing file is file $LIST.

Messages

For NOS/VE Sort/Merge, messages are written to the list and error files.

Messages are written to the dayfile for NOS Sort/Merge 5.

Owncode Procedures

For NOS/VE Sort/Merge, any owncode procedures specified for a sort or merge must be accessible from an object library in the current object library list. If you enter an owncode procedure name in lowercase letters, Sort/Merge does not convert the name to uppercase letters. Uppercase letters must be used when naming an owncode procedure.

Procedures for NOS/VE Only

New procedures for NOS/VE Sort/Merge include: SM5DUCT, SM5LCT, and SM5LO procedure calls.

Signed Overpunches

34 overpunches are defined for NOS/VE Sort/Merge; 20 overpunches are defined for NOS Sort/Merge 5.

SM5EL Procedure

The maximum error level can only be specified as a letter for NOS/VE Sort/Merge.

SM5OWNn Procedures

For NOS/VE Sort/Merge, an owncode procedure is specified by the entry point name. If you enter the owncode routine name in lowercase letters, NOS/VE Sort/Merge will not convert the name to uppercase letters. Uppercase letters must be used to name an owncode procedure.

SM5ST Procedure

The NOS/VE SM5ST procedure specifies a status variable in which the completion status of the command or procedure is returned.

Zero Comparison

Positive and negative zero are ordered equally for NOS/VE Sort/Merge.

Negative zero is ordered before positive zero for NOS Sort/Merge 5.

Index

A

$AASM FIT keyword 7-5
Abbreviations for FIT Keywords and Values 7-3
Aborting a parcel, with PABORT 6-41
About this manual 5
Access and share modes
  About access modes 3-3
  About share modes 3-4
  $ACCESS_AND_SHARE_MODES FIT keyword 7-5
  $ACCESS_MODE FIT keyword 7-8
  Example of storing modes in a FIT 3-7, 8
  File interface calls that use access and share modes 3-6
  FIT keywords that affect access and share modes 3-5
  Glossary definition of access modes A-1
  Glossary definition of share modes A-7
  How OPENM call affects access and share modes 3-6
  How to set access and share modes 3-6
  Overview 3-3
  Set by the open_option parameter of an OPENM call 3-7
Accessing records, during a parcel 4-4
Accuracy of result sets 5-3
Actual_result_set_placement parameter
  RSBUILD call 6-51
  RSCOMB call 6-57
ADA, using keyed-file interface calls in other languages 6-4
Adding a primary-key value to a result set, RSPUT call 6-67
Adding or deleting key values from result sets 5-6
Advanced Access Methods, glossary definition A-1
Alternate index
  Description 2-2
  Fetching information from 2-18
  Glossary definition A-1
Alternate-index blocks with specified values, KLSPACE call 6-29
Alternate-key definition
  Description 2-2
  Glossary definition A-1
Alternate keys
  Characteristics 2-1
  Collated key values 2-17
  Concatenated keys 2-6
  Creating with RMKDEF 6-50
  Creatings 2-15
  Duplicate values 2-3
  Duplicate values, processing errors for 2-4
  General 2-1
  Glossary definition A-1
  Null suppression 2-4
  Repeating groups 2-7
  Selecting 2-15
  Sparse-key control 2-5
  Specifying values 2-16
  Variable lengths 2-8
$AM FIT keyword 7-8
$ARL FIT keyword 7-12
Array parameter
  SM5MERGE call 8-33
  SM5SORT call 8-44
Ascending sort order, glossary definition A-1
ASCII Character Set
  Collating Weight Tables C-1, 2
Assistance, in case you need help 9
Attributes
  Of keyed files 1-18
  Required for parcels 4-2
$AU FIT keyword 7-11
Audience, for this manual 5
$AUTOMATIC_UNLOCK
  Using IFETCH to retrieve the value 6-20
$AUTOMATIC_UNLOCK FIT keyword 7-11
Automatic_unlock parameter, LOCKK call 6-35
$AVERAGE_RECORD_LENGTH FIT keyword 7-12

B

Basic Access Methods, glossary definition A-1
BASIC, using keyed-file interface calls in other languages 6-4
Beginning a parcel, with PBEGIN 6-42
BINARY_BITS numeric data format, in Sort/Merge 8-5
BINARY numeric data format, in Sort/Merge 8-5
Block_count parameter, KLSPACE call 6-29
Block, glossary definition A-1
Block_space parameter, KLSPACE call 6-29
Building a result set, RSBUILD call 6-51
Byte-addressable file organization, glossary definition A-1

C

C, using keyed-file interface calls in other languages 6-4
Call, COLSEQ Parameter C-1
CALL, STOREF statement D-1
Calls
  Disallowed during a parcel 4-9
  In Sort/Merge 8-16.1
  Keyed-file interface calls 6-1
Char parameter, SM5SEQS call 8-43
Character key type, sort keys in Sort/Merge 8-3
Clearing a file lock, UNLOCKF call 6-78
Clearing a key lock, UNLOCKK call 6-79
Clearing expired locks 3-14
Close_flag parameter, CLOSEM call 6-6
CLOSEM call
  Description 6-6
  Disallowed during a parcel 4-9
Closing a keyed-file, CLOSEM call 6-6
Closing a result set, RSCLOSE call 6-56
COBOL, using keyed-file interface calls in other languages 6-4
$COLLATE_TABLE FIT keyword 7-13
$COLLATE_TABLE_NAME FIT keyword 7-14
$COLLATE TABLE NAME keyword D-1
Collated alternate key values 2-17
Collated keys
  General D-1
  Glossary definition A-2
Collated primary keys
  In direct-access keyed files 1-15
  In indexed-sequential keyed files 1-10
Collating sequences
  Defining your own 8-73
  Glossary definition A-2
  In Sort/Merge 8-5
  List of C-1
Collating_table_name parameter, SM5DUCT call 8-19
Collating Weight Tables
  General C-1
  OSV$ASCII6_FOLDED C-6
  OSV$ASCII6_STRICT C-8
  OSV$COBOL6_FOLDED C-10
  OSV$COBOL6_STRICT C-12
  OSV$DISPLAY63_FOLDED C-14
  OSV$DISPLAY63_STRICT C-16
  OSV$DISPLAY64_FOLDED C-18
  OSV$DISPLAY64_STRICT C-20
  OSV$EBCDIC C-22
  OSV$EBCDIC6_FOLDED C-29
  OSV$EBCDIC6_STRICT C-31
  Standard ASCII C-2
Collation table
  Creating your own D-2
  Example D-3
  Glossary definition A-2
  Specifying a predefined D-1
Collation_table_name parameter, SM5LCT call 8-30
Collation weight, glossary definition A-2
Collation weight tables, creating D-10
COLSEQ Parameter C-1
Combining result sets
  Description 5-4
  Using the RSCOMB call 6-57
Comments, how to submit 8
Committing a parcel, with PCOMMIT 6-42.2
$COMPRESSION_PROCEDURE_NAME FIT keyword 7-15
Computing home blocks, for direct-access keyed files 1-14
Concatenated keys, in alternate keys 2-6
Concepts
  About FORTRAN keyed-file interface 1-16
  About the keyed-file interface 1-1
Condition_code parameter
  KEYLIST call 6-22
  KLCOUNT call 6-26
  KLSPACE call 6-29
  PABORT call 6-41
  PBEGIN call 6-42
  PCOMMIT call 6-42.2
  PDETERM call 6-42.4
  RSBUILD call 6-51
  RSCLEAR call 6-55
  RSCLOSE call 6-56
  RSCOMB call 6-57
  RSDLTE call 6-59
  RSGETN call 6-60
  RSINFO call 6-63
  RSOPEN call 6-65
  RSPUT call 6-67
  RSSKIP call 6-69
  RSSTART call 6-70
CONDSYM, using CONDSYM to process errors 6-3
Conflict tables, for locks 3-17
Conventions, of this manual 7
Converting programs from FORTRAN 5 to NOS/VE FORTRAN E-1
Count parameter
  RSSKIP call 6-69
  SKIP call 6-71
Counting alternate-index blocks with specified values, KLSPACE call 6-29
Counting the number of primary-key values, KLCOUNT call 6-26
$CPN FIT keyword 7-15
Creating a collation table D-1
Creating a collation weight table D-10

Creating a FIT
For a direct-access file, FILEDA call 6-9
For an indexed-sequential file, FILEIS call 6-10
Creating a keyed file 1-17
Creating alternate keys
General 2-15
Using a RMKDEF call 6-50
Creating an object library 8-70
$CT FIT keyword 7-13
$CTN FIT keyword 7-14
CYBER Record Manager (AAM) subprograms E-1
CYBER Software Support 9

D

Data blocks
Data-block split
Glossary definition A-2
In an indexed-sequential keyed file 1-6
Glossary definition A-2
Padding, glossary definition A-2
Data Exit Procedure FIT keyword 7-19
Data_file parameter, RSOPEN call 6-65
$DATA_PADDING FIT keyword 7-17
Data records, in indexed-sequential keyed files 1-4
Days parameter, PDETERM call 6-42.4
DCT Keyword D-1
$DD FIT keyword 7-18
Deadlock, when using locks 3-16
Debug for NOS/VE manual B-2
Defining an alternate key 2-2
Defining sort keys, in Sort/Merge 8-3
Defining your own collating sequence 8-73
$DELETE_DATA FIT keyword 7-18
Deleting a primary-key value from a result set, RSDLTE call 6-59
Deleting a record, DLTE call 6-7
Deleting (or adding) key values from result sets 5-6
Descending sort order, glossary definition A-2
Determine_from_access_modes, a special access and share mode 3-4
Diagnostic Messages manual, looking up condition names 6-3
Differences between NOS/VE FORTRAN and FORTRAN 5 E-1
Direct access input/output, glossary definition A-2
Direct-access keyed files
Computing home blocks 1-14
Creating a FIT with FILEDA 6-9
General 1-11
Hashing procedures 1-14
Home blocks 1-12
Overflow block 1-13
Primary keys 1-15
Direction parameter, PDETERM call 6-42.4
Discarding existing result set, RSCLEAR call 6-55
DLTE call 6-7
$DP FIT keyword 7-17
Duplicate key value control, glossary definition A-2
Duplicate key values
Glossary definition A-2
In alternate keys 2-3
Processing errors 2-4
DX
FIT keyword 7-19
Using IFETCH to retrieve the value 6-20

E

$EC FIT keyword 7-21
$EEN FIT keyword 7-22
$EEP FIT keyword 7-23
$EEPN FIT keyword 7-22
$EK FIT keyword 7-20
$EL FIT keyword 7-24
$EMBEDDED_KEY FIT keyword 7-20
Embedded key, glossary definition A-3
End_of_primary_key_list parameter, KEYLIST call 6-22
$ERC FIT keyword 7-27
$ERROR_COUNT FIT keyword 7-21
$ERROR_EXIT_NAME FIT keyword 7-22
$ERROR_EXIT_PROCEDURE FIT keyword 7-23
$ERROR_EXIT_PROCEDURE_NAME FIT keyword 7-22
Error_exit_procedure parameter
DLTE call 6-7
GET call 6-13
GETN call 6-17
PUT call 6-43
PUTREP call 6-45
REPLC call 6-47
STARTM call 6-74
Error-exit procedures, in the FORTRAN keyed-file interface 1-21
$ERROR_LIMIT FIT keyword 7-24
$ERROR_STATUS
FIT keyword 7-25
Using $ERROR_STATUS to process errors 6-3
Using IFETCH to retrieve the value 6-20
Using to process errors 1-20

Errors
  Processing errors for keyed-file interface calls 6-3
  Recovering from result set errors 5-4
$ES FIT keyword 7-25
$ESTIMATED_RECORD_COUNT FIT keyword 7-27
Example
  Of using sort/merge 8-64, 68
  Parcel program 4-11
  Storing access and share modes in a FIT 3-7, 8
Exception processing
  in Sort/Merge 8-11
  Invalid key data 8-14
  Summing errors 8-14
Exclusive_access lock intent
  File lock 3-12
  Key lock 3-11
Expired locks
  Description 3-14
  During a parcel 4-6

F

F Record type, glossary definition A-3
Fatal/Nonfatal Flag FIT keyword 7-32
Fetching, from the alternate index 2-18
Fetching primary-key values from alternate index, KEYLIST call 6-22
$FI FIT keyword 7-28
$FILE_IDENTIFIER FIT keyword 7-28
File Information Table
  Description 1-16
  Glossary definition A-3
  Keywords and values 7-3
File interface calls that use access and share modes 3-6
File-level parcels
  Description 4-10
  Glossary definition A-3
File locks
  Clearing with UNLOCKF 6-78
  Description 3-12
  Requesting with LOCKF 6-33
$FILE_ORGANIZATION
  Using IFETCH to retrieve the value 6-20
$FILE_ORGANIZATION FIT keyword 7-29
File organization, glossary definition A-3
File parameter
  SM5E call 8-20
  SM5ERF call 8-24
  SM5FROM call 8-26
  SM5LIST call 8-31
  SM5TO call 8-48
$FILE_POSITION
  FIT keyword 7-30
  Using IFETCH to retrieve the value 6-20
File_position parameter
  KEYLIST call 6-22
  OPENM call 6-38
File-spanning parcel
  Glossary definition A-3
  Using PDETERM to retrieve information about 6-42.3
FILEDA call 6-9
FILEIS call 6-10
Files
  Characteristics of sort/merge files 8-16.2
  Result set 5-4
First parameter
  SM5KEY call 8-27
  SM5SUM call 8-46
First_result_set parameter, RSCOMB call 6-57
FIT
  Description 1-16
  FIT values affecting parcels 4-5
FIT, also see File Information Table A-3
Fit_keyword parameter
  FILEDA call 6-9
  FILEIS call 6-10
  IFETCH call 6-20
  STOREF call 6-76
FIT Keywords 7-3
Fit_list parameter, PBEGIN call 6-42
Fit_value parameter
  FILEDA call 6-9
  FILEIS call 6-10
  STOREF call 6-76
Fixed_length parameter, SM5OFL call 8-35
FLUSHM call 6-12
FNF
  FIT keyword 7-32
  Using IFETCH to retrieve the value 6-20
$FO FIT keyword 7-29
For better performance
  Specify exclusive access to a file 3-2
  Specify minimal share modes to a file 3-2
  Using file-level parcels 4-10
  Using parcel logs 4-7
$FORCED_WRITE FIT keyword 7-31
FORTRAN keyed-file interface
  Concepts 1-16
  Processing errors 1-20
  Using error-exit procedures 1-21
FORTRAN keyed-file interface calls disallowed during a parcel 4-9

FORTRAN manuals
FORTRAN for NOS/VE Summary B-2
FORTRAN for NOS/VE Topics for FORTRAN Programmers B-2
FORTRAN for NOS/VE Tutorial B-2
FORTRAN manual set description 6
FORTRAN Version 1 for NOS/VE Language Definition Usage B-2
FORTRAN Version 2 for NOS/VE Language Definition Usage B-2
FORTRAN programs, how to use result sets in 5-2
FORTRAN 5, converting FORTRAN 5 programs to NOS/VE FORTRAN E-1
$FP FIT keyword 7-30
$FW FIT keyword 7-31

G

$GAL FIT keyword 7-33
$GAM FIT keyword 7-34
$GET_AND_LOCK
FIT keyword 7-33
Using IFETCH to retrieve the value 6-20
GET call 6-13
GETN call 6-17
$GLOBAL_ACCESS_MODES
FIT keyword 7-34
Using IFETCH to retrieve the value 6-20
$GLOBAL_SHARE_MODES
FIT keyword 7-35
Using IFETCH to retrieve the value 6-20
$GSM FIT keyword 7-35

H

Hashing procedure
Glossary definition A-3
In direct-access keyed files 1-14
$HASHING_PROCEDURE_NAME FIT keyword 7-36
Help, from CYBER Software Support 9
High_key parameter
KEYLIST call 6-22
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
High_key_relation parameter
KEYLIST call 6-22
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
Home blocks
Computing for direct-access keyed files 1-14
For direct-access keyed files 1-12
Glossary definition A-3
Hotline, to CYBER Software Support 9
Hours parameter, PDETERM call 6-42.4
How to use FIT keywords in FORTRAN programs 7-3
$HPN FIT keyword 7-36

I

IFETCH
Description of IFETCH call 6-20
Using IFETCH to process errors 6-3
$IHBC FIT keyword 7-39
$IL FIT keyword 7-37
In case you need assistance 9
In-memory sort, example of 8-68
Index, alternate 2-2
Index blocks
Alternate-index blocks with specified values, KLSPACE call 6-29
For indexed-sequential keyed files 1-4
Glossary definition A-3
Index-block padding, glossary definition A-3
Index-block split, glossary definition A-4
Index levels
Glossary definition A-4
In indexed-sequential keyed files 1-8
Index level overflow, glossary definition A-4
$INDEX_LEVELS FIT keyword 7-37
$INDEX_PADDING FIT keyword 7-38
Index record A-4
Indexed-sequential file
Creating a FIT with FILEIS 6-10
Data-block split 1-6
Data records 1-4
General 1-3
Index blocks 1-4
Index levels 1-8
Indexed-sequential file organization, glossary definition A-4
Internal tables 1-4
Primary key types 1-10
Primary keys 1-10
Reading an indexed-sequential file 1-5
$INITIAL_HOME_BLOCK_COUNT FIT keyword 7-39
Input and output files, in sort/merge 8-16.2
Instance of open
Description 3-1
Glossary definition A-4
INTEGER_BITS numeric data format, in Sort/Merge 8-5
Integer key, glossary definition A-4
INTEGER numeric data format, in Sort/Merge 8-5

Integer primary keys
  In direct-access keyed files 1-15
  In indexed-sequential keyed files 1-10
Internal tables, for indexed-sequential keyed files 1-4
Invalid key data, in Sort/Merge 8-14
Invalid records, in Sort/Merge 8-13
$IP FIT keyword 7-38

J

Job, glossary definition A-4

K

$KA FIT keyword 7-40
$KEY_ADDRESS
  FIT keyword 7-40
  Using IFETCH to retrieve the value 6-20
Key_area parameter
  DLTE call 6-7
  GET call 6-13
  GETN call 6-17
  LOCKK call 6-35
  PUT call 6-43
  PUTREP call 6-45
  REPLC call 6-47
  RSGETN call 6-60
  STARTM call 6-74
  UNLOCKK call 6-79
Key_count parameter, RSINFO call 6-63
Key, glossary definition A-4
$KEY_LENGTH FIT keyword 7-41
Key length, sort keys in Sort/Merge 8-3
Key list, glossary definition A-4
Key_location parameter
  RSDLTE call 6-59
  RSPUT call 6-67
  RSSTART call 6-70
Key locks, clearing with UNLOCKK 6-79
$KEY_NAME FIT keyword 7-42
$KEY_POSITION
  FIT keyword 7-43
  Using IFETCH to retrieve the value 6-20
Key position, sort keys in Sort/Merge 8-3
$KEY_RELATION
  FIT keyword 7-44
  Using IFETCH to retrieve the value 6-20
Key_relation parameter, RSSTART call 6-70
Key-Type attribute D-1
$KEY_TYPE FIT keyword 7-45
Key_type parameter
  SM5DUCT call 8-19
  SM5KEY call 8-27
  SM5LCT call 8-30
Key types
  Glossary definition A-5
  In Sort/Merge 8-4
Key values, adding or deleting from result sets 5-6
Keyed file
  Attributes 1-18
  Collated keys in keyed files D-1
  Creating one with the FORTRAN keyed-file interface 1-17
  Definition 1-2
  Organization 1-2
  Organization, glossary definition A-5
  Positioning using a key value with STARTM 6-74
  Repositioning with SKIP 6-71
  Sharing 3-1
  Storing a value in a FIT with STOREF 6-76
  Using an existing one with the FORTRAN keyed-file interface 1-19
Keyed-File Interface Calls
  General 6-1
  Processing errors 6-3
  Summary of calls 6-2
Keyed-file interface concepts, general 1-2
KEYLIST call
  Description 6-22
  Disallowed during a parcel 4-9
Keys, alternate
  Characteristics 2-1
  General 2-1
Keys, for sorts, in Sort/Merge 8-2
Keys_remaining parameter, RSINFO call 6-63
Keyword, glossary definition A-5
Keyword parameter, SM5ZLR 8-51
$KL FIT keyword 7-41
KLCOUNT call
  Description 6-26
  Disallowed during a parcel 4-9
KLSPACE call 6-29
$KN FIT keyword 7-42
$KP FIT keyword 7-43
$KR FIT keyword 7-44
$KT FIT keyword 7-45

L

$LAST_OPERATION
  FIT keyword 7-46
  Using IFETCH to retrieve the value 6-20

Length parameter
SM5KEY call 8-27
SM5SUM call 8-46
$LET FIT keyword 7-48
$LFN FIT keyword 7-47
$LI FIT keyword 7-49
LISP, using keyed-file interface calls in other languages 6-4
List_count_limit parameter, KLCOUNT call 6-26
List_count parameter, KLCOUNT call 6-26
$LO FIT keyword 7-46
$LOCAL_FILE_NAME
FIT keyword 7-47
Using IFETCH to retrieve the value 6-20
$LOCK_EXPIRATION_TIME FIT keyword 7-48
$LOCK_INTENT
FIT keyword 7-49
Using IFETCH to retrieve the value 6-20
Lock_intent parameter
LOCKF call 6-33
LOCKK call 6-35
Lock manager, NOS/VE 3-9
LOCKF call
Description 6-33
Disallowed during a parcel 4-9
LOCKK call 6-35
Locks
Conflict tables 3-17
Deadlock 3-16
Definition 3-9
Example 3-10
Expiration and clearing 3-16
Expiration during a parcel 4-6
File locks 3-12
Glossary definition A-5
Lock intents 3-11
Lock renewal 3-12
Reasons for 3-9
Sharing keyed files 3-9
Waiting for a lock 3-13
When does sharing keyed files require locks? 3-2
Log, glossary definition A-5
Log parameter, PDETERM call 6-42.4
$LOG_RESIDENCE FIT keyword 7-50
$LOGGING_OPTIONS FIT keyword 7-51
Logical_relation parameter
RSBUILD call 6-51
RSCOMB call 6-57
Low_key parameter
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
Low_key_relation parameter
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
$LR FIT keyword 7-50

M

Major_high_key parameter
KEYLIST call 6-22
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
$MAJOR_KEY_LENGTH
FIT keyword 7-52
Using IFETCH to retrieve the value 6-20
Major_key_length parameter
GET call 6-13
RSSTART call 6-70
STARTM call 6-74
Major_low_key parameter
KLCOUNT call 6-26
KLSPACE call 6-29
RSBUILD call 6-51
Major sort key, glossary definition A-5
Manual history 2
Manual set, FORTRAN 6
Manuals, how to order 8
Math Library for NOS/VE B-2
Max_record_length parameter
SM5FMA call 8-25
SM5MA call 8-47
$MAXBL FIT keyword 7-53
$MAXIMUM_BLOCK_LENGTH FIT keyword 7-53
Maximum_length parameter, SM5OMRL call 8-37
$MAXIMUM_RECORD_LENGTH FIT keyword 7-54
$MAXRL FIT keyword 7-54
$MC FIT keyword 7-55
Merge, glossary definition A-5
Message_area parameter, PDETERM call 6-42.4
$MESSAGE_CONTROL FIT keyword 7-55
Messages manual, looking up condition names 6-3
Migration from NOS to NOS/VE B-2
$MINIMUM_RECORD_LENGTH FIT keyword 7-56
Minor sort key, glossary definition A-5
$MINRL FIT keyword 7-56
Minutes parameter, PDETERM call 6-42.4
$MKL FIT keyword 7-52
Modify share mode, when to specify 3-2
Multiple sort keys, in Sort/Merge 8-2

N

Name parameter
  SM5OWN call 8-38
  SM5SEQN call 8-41
Name_type parameter, PDETERM call 6-42.3
$NESTED_FILE_NAME FIT keyword 7-57
Nested_file parameter, RSOPEN call 6-65
Nested files 2-13
New_result_placement parameter
  RSBUILD call 6-51
  RSCOMB call 6-57
Next_key parameter, RSINFO call 6-63
$NFN FIT keyword 7-57
Nonembedded key, glossary definition A-5
NOS/VE FORTRAN and FORTRAN 5 differences E-1
NOS/VE lock manager 3-9
NOS/VE Manuals
  Description B-2
  NOS/VE Advanced File Management Usage B-2
  NOS/VE Commands and Functions B-2
  NOS/VE Diagnostic Messages B-2
  NOS/VE Object Code Management Usage B-2
  NOS/VE Source Code Management Usage B-2
  NOS/VE System Usage B-2
Null suppression, in alternate keys 2-4
Number_of_fits parameter, PBEGIN call 6-42
Number_of_records parameter, SM5FMA call 8-25
NUMERIC_FS
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-5
NUMERIC_LO
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-6
NUMERIC_LS
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-6
NUMERIC_NS
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-6
NUMERIC_TO
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-6
NUMERIC_TS
  Key type, sort keys in Sort/Merge 8-3
  Numeric data format, in Sort/Merge 8-6

O

Object library
  Adding the sort/merge object library 8-16.1
  Creating 8-70
OC
  FIT keyword 7-58
  Using IFETCH to retrieve the value 6-20
Old/New Flag FIT keyword 7-59
ON
  FIT keyword 7-59
  Using IFETCH to retrieve the value 6-20
$OP FIT keyword 7-60
Open/Close Flag FIT keyword 7-58
Open_option parameter, OPENM call 6-38
$OPEN_POSITION FIT keyword 7-60
$OPEN_SHARE_MODES FIT keyword 7-61
Open share modes, glossary definition A-6
Opening a keyed file with OPENM 6-38
Opening a result set, RSOPEN call 6-65
OPENM call
  Description 6-38
  How it affects access and share modes 3-6
  How the open_option parameter sets access and share modes 3-7
Option parameter
  SM5CC call 8-18
  SM5LO call 8-32
  SM5OMIT call 8-36
  SM5RETA call 8-39
  SM5SEQA call 8-40
  SM5SEQR call 8-42
  SM5VER call 8-50
Order parameter, SM5KEY call 8-27
Ordering manuals 8
Organization
  Of direct-access keyed files 1-11
  Of indexed-sequential keyed files 1-3
  Of keyed files 1-2
  Of this manual 6
$OSM FIT keyword 7-61
OSV$ Collating Weight Tables
  OSV$ASCII6_FOLDED C-6
  OSV$ASCII6_STRICT C-8
  OSV$COBOL6_FOLDED C-10
  OSV$COBOL6_STRICT C-12
  OSV$DISPLAY63_FOLDED C-14
  OSV$DISPLAY63_STRICT C-16

OSV$DISPLAY64_FOLDED C-18
OSV$DISPLAY64_STRICT C-20
OSV$EBCDIC C-22
OSV$EBCDIC6_FOLDED C-29
OSV$EBCDIC6_STRICT C-31
Overflow blocks
For direct-access keyed files 1-13
Glossary definition A-6
Owncode, glossary definition A-6
Owncode procedures, writing your own for Sort/Merge 8-52
Owncode1, in Sort/Merge 8-56
Owncode2, in Sort/Merge 8-57
Owncode3, in Sort/Merge 8-58
Owncode4, in Sort/Merge 8-59
Owncode5, in Sort/Merge 8-60

P

$P FIT keyword 7-63
PABORT call 6-41
PACKED_NS numeric data format, in Sort/Merge 8-7
PACKED numeric data format, in Sort/Merge 8-7
Padding, glossary definition A-6
Page aging interval, in Sort/Merge 8-16
Parcel log
Glossary definition A-6
Using 4-7
Parcel_name parameter, PDETERM call 6-42.3
Parcel states, returned by PDETERM 4-8
Parcels
Aborting with PABORT 6-41
Beginning with PBEGIN 6-42
Calls disallowed during a parcel 4-9
Committing with PCOMMIT 6-42.2
File-spanning parcels 4-1
FIT values affecting parcels 4-5
Glossary definition A-6
How to use parcels 4-2
Lock expiration during a parcel 4-6
Parcel processing outline 4-2
Parcel states returned by PDETERM 4-8
Program example 4-11
Record access during a parcel 4-4
Required attributes for parcels 4-2
Using PDETERM for a file-spanning parcel 6-42.3
Using the parcel log 4-7
What are parcels? 4-1
Partial numeric sum fields, in Sort/Merge 8-11
Partial sum fields, in Sort/Merge 8-11
Partition, glossary definition A-6
Pascal, using keyed-file interface calls in other languages 6-4
$PASSWORD FIT keyword 7-63
PBEGIN call
Description 6-42
Disallowed during a parcel 4-9
PCOMMIT call 6-42.2
PDETERM call 6-42.3
Performance considerations in Sort/Merge 8-15
Permitted_access_modes, a special access and share mode 3-4
$PKA FIT keyword 7-64
Position parameter, RSINFO call 6-63
Positions a keyed file using a key value, STARTM call 6-74
Positions a result set using a primary-key value, RSSTART call 6-70
Preserve_access_and_content lock intent
File lock 3-13
Key lock 3-12
Preserve_content lock intent
File lock 3-13
Key lock 3-12
Previous_key parameter, RSINFO call 6-63
$PRIMARY_KEY_ADDRESS
FIT keyword 7-64
Using IFETCH to retrieve the value 6-20
Primary keys
Glossary definition A-6
In direct-access keyed files 1-15
In indexed-sequential keyed files 1-10
Types, in indexed-sequential keyed-files 1-10
Values
Counting number of primary-key values, KLCOUNT call 6-26
Fetching from alternate index, KEYLIST call 6-22
Lock, requesting with LOCKK 6-35
Procedure calls in Sort/Merge 8-16.1
Processing errors for duplicate values in alternate keys 2-4
Processing errors, in the FORTRAN keyed-file interface 1-20
Professional Programming Environment for NOS/VE B-2
Programming Environment for NOS/VE B-2
PROLOG, using keyed-file interface calls in other languages 6-4
PUT call 6-43
PUTREP call 6-45

R

Random access, glossary definition A-6
Random file organization, glossary definition A-6
Reading
  A record by key value, GET call 6-13
  A record using a result set, RSGETN call 6-60
  An indexed-sequential keyed file 1-5
  Current information about a result set, RSINFO call 6-63
  The next record, GETN call 6-17
REAL numeric data format, in Sort/Merge 8-7
Record access, during a parcel 4-4
Record length
  FIT keyword 7-65
  In Sort/Merge 8-10
Record_length parameter
  PUT call 6-43
  PUTREP call 6-45
  REPLC call 6-47
$RECORD_LIMIT FIT keyword 7-66
Record type F, glossary definition A-3
$RECORD_TYPE FIT keyword 7-67
$RECORDS_PER_BLOCK FIT keyword 7-68
Recovery, glossary definition A-7
Renewal, locks 3-12
Repeat parameter, SM5SUM call 8-46
Repeating groups
  Glossary definition A-7
  In alternate keys 2-7
Replacing a record
  PUTREP call 6-45
  REPLC call 6-47
REPLC call 6-47
Repositions a keyed file, SKIP call 6-71
Repositions a result set, RSSKIP call 6-69
Requesting a file lock, LOCKF call 6-33
Requesting a primary-key value lock, LOCKK call 6-35
Required_share_modes, a special access and share mode 3-4
Result_set_file parameter
  RSOPEN call 6-65
Result_set_id parameter
  RSCLEAR call 6-55
  RSCLOSE call 6-56
  RSINFO call 6-63
  RSOPEN call 6-65
Result_set_not parameter
  RSGETN call 6-60
Result sets
  Adding a primary-key value with RSPUT 6-67
  Adding or deleting key values 5-6
  Building with RSBUILD 6-51
  Closing with RSCLOSE 6-56
  Combining result sets 5-4
  Combining with RSCOMB 6-57
  Deleting a primary-key value with RSDLTE 6-59
  Discarding current with RSCLEAR 6-55
  Files 5-4
  Keeping accurate 5-3
  Opening with RSOPEN 6-65
  Outline of how to use result sets 5-2
  Positioning using a primary-key value with RSSTART 6-70
  Reading a record with RSGETN 6-60
  Reading current information with RSINFO 6-63
  Recovering from read errors 5-4
  Repositioning with RSSKIP 6-69
  Validity 5-3
  What are result sets? 5-1
Returned_length parameter, PDETERM call 6-42.5
Rewinding a file, REWND call 6-49
REWND call 6-49
$RL FIT keyword 7-66
RL FIT keyword 7-65
RMKDEF call
  Description 6-50
  Disallowed during a parcel 4-9
$RPB FIT keyword 7-68
RSBUILD call
  Description 6-51
  Disallowed during a parcel 4-9
RSCLEAR call 6-55
RSCLOSE call 6-56
RSCOMB call
  Description 6-57
  Disallowed during a parcel 4-9
RSDLTE call
  Description 6-59
  Disallowed during a parcel 4-9
RSGETN call 6-60
RSINFO call 6-63
RSOPEN call 6-65
RSPUT call
  Description 6-67
  Disallowed during a parcel 4-9
RSSKIP call 6-69
RSSTART call 6-70
$RT FIT keyword 7-67

S

$SC FIT keyword 7-69
Second_result_set parameter, RSCOMB call 6-57
Selecting alternate keys 2-15
Sequential access, glossary definition A-7
Sequential file organization, glossary definition A-7

Severity_level parameter
SM5EL call 8-21
SM5ST call 8-45
Share modes
Description 3-4
Glossary definition A-7
Sharing keyed files
Description 3-1
Using access and share modes 3-5
When does sharing require locks? 3-2
Sign overpunch table, for Sort/Merge 8-9
SKIP call 6-71
$SKIP_COUNT FIT keyword 7-69
SM5 Sort/Merge procedure calls
SM5CC 8-18
SM5DUCT 8-19
SM5E 8-20
SM5EL 8-21
SM5END 8-22
SM5ENR 8-23
SM5ERF 8-24
SM5FMA 8-25
SM5FROM 8-26
SM5KEY 8-27
SM5LCT 8-30
SM5LIST 8-31
SM5LO 8-32
SM5MERG 8-33
SM5OFL 8-35
SM5OMIT 8-36
SM5OMRL 8-37
SM5OWN 8-38
SM5RETA 8-39
SM5SEQA 8-40
SM5SEQN 8-41
SM5SEQR 8-42
SM5SEQS 8-43
SM5SORT 8-44
SM5ST 8-45
SM5SUM 8-46
SM5TMA 8-47
SM5TO 8-48
SM5VER 8-50
SM5ZLR 8-51
Sort, glossary definition A-7
Sort keys
Glossary definition A-7
In Sort/Merge 8-2
Sort/Merge
Adding the sort/merge object library 8-16.1
Collating sequences 8-5
Example 8-64, 68
General 8-1
Input and output files 8-16.2
Invalid records 8-13
Key types 8-4
Limiting memory usagee 8-15
Page aging interval 8-16
Performance considerations 8-15
Procedure calls 8-16.1
Record length 8-10
Sort keys 8-2
Sort order 8-9
Summing records 8-72
Zero-length records 8-12
Sort order
Glossary definition A-7
In Sort/Merge 8-9
Source_result_set parameter
RSBUILD call 6-51
RSGETN call 6-60
RSSKIP call 6-69
RSSTART call 6-70
Sparse-key control
Glossary definition A-7
In alternate keys 2-5
Specifying alternate-key values 2-16
STARTM call 6-74
State parameter, PDETERM call 6-42.3
STOREF
Call 6-76
Statement D-1
Storing a value in a FIT, STOREF call 6-76
Submitting comments 8
Subprograms, CYBER Record Manager E-1
Summing errors, in Sort/Merge 8-14
Summing records, in Sort/Merge 8-72
System_parcel_name parameter
PABORT call 6-41
PBEGIN call 6-42
PCOMMIT call 6-42.2

T

Target_result_set parameter
RSBUILD call 6-51
RSCOMB call 6-57
RSDLTE call 6-59
RSPUT call 6-67
Task, glossary definition A-7
Transferred_byte_count parameter, KEYLIST call 6-22
Transferred_key_count parameter, KEYLIST call 6-22
Type parameter, SM5SUM call 8-46

U

U record type, glossary definition A-8
Uncollated keys
Glossary definition A-8
In direct-access keyed files 1-15
In indexed-sequential keyed files 1-10
UNLOCKF call
Description 6-78
Disallowed during a parcel 4-9
UNLOCKK call 6-79

Update recovery log, glossary definition A-8
User_parcel_name parameter, PBEGIN call 6-42
Using access and share modes, to share a file 3-5
Using an existing keyed file 1-19

V

V record type, glossary definition A-8
Validity of result sets 5-3
Value parameter, SM5ENR call 8-23
Variable-lengths, in alternate keys 2-8
Variable parameter
  IFETCH call 6-20
  SM5FMA call 8-25
  SM5MA call 8-47

W

$WAIT_FOR_ATTACHMENT FIT keyword 7-70
$WAIT_FOR_LOCK
  FIT keyword 7-71
  Using IFETCH to retrieve the value 6-20
Wait_for_lock parameter
  LOCKF call 6-33
  LOCKK call 6-35
Waiting for a lock 3-13
$WFA FIT keyword 7-70
$WFL FIT keyword 7-71
$WORKING_STORAGE_ADDRESS
  FIT keyword 7-72
  Using IFETCH to retrieve the value 6-20
Working_storage_area parameter
  GET call 6-13
  GETN call 6-17
  KEYLIST call 6-22
  PUT call 6-43
  PUTREP call 6-45
  REPLC call 6-47
  RSGETN call 6-60
$WORKING_STORAGE_LENGTH
  FIT keyword 7-73
  Using IFETCH to retrieve the value 6-20
Working_storage_length parameter
  KEYLIST call 6-22
Writing a record
  PUT call 6-43
  PUTREP call 6-45
Writing modified blocks to a file, FLUSHM call 6-12
Writing owncode procedures, for Sort/Merge 8-52
$WSA FIT keyword 7-72
$WSL FIT keyword 7-73

Z

Zero-length records, in Sort/Merge 8-12

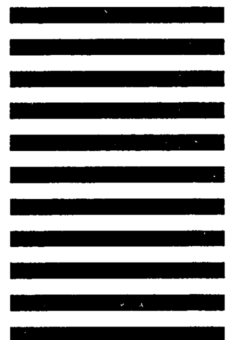
Comments (continued from other side)

Please fold on dotted line;
seal edges with tape only.

FOLD

FOLD

FOLD



BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technical Publications
SVLF45
5101 Patrick Henry Drive
Santa Clara, CA 95054-1111

FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces 60485917 B

We would like your comments on this manual to help us improve it. Please take a few minutes to fill out this form.

Who are you?

- ☐ Manager
- ☐ Systems analyst or programmer
- ☐ Applications programmer
- ☐ Operator
- ☐ Other ____________

How do you use this manual?

- ☐ As an overview
- ☐ To learn the product or system
- ☐ For comprehensive reference
- ☐ For quick look-up
- ☐ Other ____________

What programming languages do you use? ____________

How do you like this manual? Answer the questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Do you use this manual frequently?

Comments? If applicable, note page and paragraph. Use other side if needed.

Check here if you want a reply: ☐

Name ____________ Company ____________

Address ____________ Date ____________

Phone ____________

Please send program listing and output if applicable to your comment.

CONTROL DATA