IBM

Series/1

SC34-0404-0

File No. S1-32

LICENSED PROGRAM

IBM Series/1 Event Driven Executive Indexed Access Method Version 2 Guide

Program Numbers: 5719-AM4
5719-UT5

IBM

Series/1

SC34-0404-0

File No. S1-32

LICENSED PROGRAM

IBM Series/1 Event Driven Executive Indexed Access Method Version 2 Guide

Program Numbers: 5719-AM4
5719-UT5

First Edition (July 1981)

This edition applies to the Event Driven Executive Indexed Access Method Version 2 Modification Level 0: Program Number 5719-AM4, until otherwise indicated by new editions or technical newsletters.

PREFACE

HOW TO USE THIS BOOK

The material in this section is a guide to using this book. It defines the purpose, audience, and content of the book as well as listing aids for using the book and background materials.

PURPOSE

The purpose of this publication is to describe how to use the Indexed Access Method Version 2 to set up indexed files and to develop application programs using indexed files.

AUDIENCE

This manual is intended for use by:

- Application designers to design applications which use the Indexed Access Method Version 2.
- Application programmers to develop applications which use the Indexed Access Method Version 2.

Applications for the Series/1 can be developed in several languages. Unless otherwise noted in this section, material in this book is intended for use in the development of applications in any of the following languages:

- COBOL
- EDL
- PL/I

HOW THIS BOOK IS ORGANIZED

This book describes the Indexed Access Method in the following order:

- Chapter 1, "Introduction" provides an overview of the Indexed Access Method.
- Chapter 2, "Using the Indexed Access Method," provides a brief description of what indexed files are, how to set up an indexed file, and application program request statements.

- Chapter 3, "Defining Primary Index Files," describes the format of the primary index file and how to use the $IAMUT1 utility to set up your indexed files.

- Chapter 4, "Loading the Primary Index File," describes loading data records into a primary index file using an application program.

- Chapter 5, "Building a Secondary Index," provides information on using secondary keys, what a secondary index is and does, and how to set up and load a secondary index.

- Chapter 6, "Processing the Indexed File," describes how to process the indexed file with an application program.

- Chapter 7, "Coding The Indexed Access Method Requests," provides information needed to code EDL applications which use the Indexed Access Method. This chapter is intended only for EDL application developers.

- Chapter 8, "The $IAMUT1 Utility," provides information needed to use $IAMUT1, including the completion codes it generates.

- Chapter 9, "The $VERIFY Utility," provides information needed to use $VERIFY.

- Chapter 10, "Storage and Performance Considerations" describes the storage and performance characteristics of the Indexed Access Method and how to tailor the Indexed Access Method to the processing requirements of your installation.

- Chapter 11, "Error Recovery" describe some of the error recovery procedures available for use with Indexed Access Method applications.

- Chapter 12, "Installing the Indexed Access Method," provides an overview of the installation process.

- Appendix A, "Summary of Calculations," provides a summary of calculations for calculating the various blocks which make up indexed files.

- Appendix B, "Preparing Indexed Access Method Programs," provides an overview of preparing an Indexed Access Method application and a sample $JOBUTIL procedure for an EDL application.

- Appendix C, "Coding Examples," provides comprehensive examples of Indexed Access Method programs. This appendix is for application developers using EDL, COBOL, or PL/I as their application programming language.

AIDS IN USING THIS PUBLICATION

Illustrations in this book are enclosed in boxes. Many illustrations display screens generated while using the Event Driven Executive system. In those cases where the actual data exceeds the size of the box, the information may be illustrated in a modified format.

In display screens appearing in this manual, operator input is shown in bold type. This highlighting is for illustrative purposes only, to distinguish data entered by the operator from that generated by the system.

Several other aids are provided to assist you in using this book:

- A bibliography
  - Lists the books in the Event Driven Executive library along with a brief description of each book
  - Lists related publications and materials
- A glossary defines terms and abbreviations
- A common index includes entries from each book in the Event Driven Executive library

RELATED PUBLICATIONS

Related publications are listed in the Bibliography.

SUBMITTING AN APAR

If you have a problem with the Series/1 Event Driven Executive services, you are encouraged to fill out an authorized program analysis report (APAR) form as described in the IBM Series/1 Authorized Program Analysis Report (APAR) User's Guide, GC34-0099.

CONTENTS

FIGURES

CHAPTER 1. INTRODUCTION

The Indexed Access Method licensed program is a data management facility that executes on an IBM Series/1 processor under the Event Driven Executive Supervisor and Emulator, Version 3.1 or later. The Indexed Access Method provides keyed access to each of your individual data records.

WHAT THE INDEXED ACCESS METHOD DOES

This licensed program builds, maintains, and accesses a data structure called an indexed file

Your data records can be loaded by the Indexed Access Method utility, $IAMUT1, or they can be loaded using an application program. Data records can then be added, deleted, modified, or accessed quickly and efficiently for processing by your application program. When reorganization of an indexed file is required the utility can be used to unload and reorganize the file.

When this licensed program is used, each of your records is identified by the contents of a predefined field called a key. The Indexed Access Method builds and maintains an index for those keys and through this index fast access to each record is provided. Your data records can be accessed either by key, or sequentially in ascending key sequence, using Indexed Access Method requests.

INDEXED ACCESS METHOD FEATURES

The Indexed Access Method offers the following features:

- Record access by a primary key or secondary keys - You can access records in an indexed file by one or more keys. Secondary keys use a separate index and Indexed Access Method provides the connection between the primary index files and seconary indexes. Duplication of secondary key fields is permitted.

- Support for high insert and delete activity - Free space can be distributed throughout the file and in a free-pool at the end of the file so that new records can be inserted. The space occupied by a deleted record is immediately available for inserting a new record.

- Direct and sequential access - You can access records either randomly by key, or sequentially in ascending key sequence.

- Data paging - You can improve Indexed Access Method performance by using data paging. With this feature active, the Indexed Access Method retains recently-used blocks of data records resident in main storage.

- Dynamic file structure - A dynamic file structure adjusts itself as needed to handle record additions and deletions. This provides a quick and easy method of designing an indexed file.

- Concurrent access to a single file by several requests - These requests can be from one or more programs. Data integrity is maintained by a file-, block-, and record-level locking system that prevents other programs from accessing the portion of the file being modified.

- Implementation as a separate task - A single copy of the Indexed Access Method executes and coordinates all requests. A buffer pool supports all requests and optimizes the space required for physical I/O; the only buffer required in an application program is the one for the record being processed.

- Input records - Either blocked or unblocked input records are accepted.

- $IAMUT1 - A utility program that allows you to maintain a secondary index directory, create, format, load, unload, and reorganize an indexed file. The load and unload functions accept either blocked or unblocked records.

- $VERIFY - A utility program that allows you to check the integrity of the index structure, print control blocks, and print a free space report for an indexed file.

- Error logging - If multiple error return codes occur, errors are logged in the system error log.

- $ILOG - The error log entries can be printed by using the $ILOG utility.

- File compatibility - Files created by the Event Driven Executive Indexed Access Method are compatible with those created by the IBM Series/1 Realtime Programming System Indexed Access Method licensed program, 5719-AM1 and 5719-AM2 provided that the block size is a multiple of 256.

- Data protection - All input/output operations are performed by system functions. Therefore, all data protection facilities offered by the system also apply to indexed files. The following additional data protection is provided:

  - The exclusive option specifies that the file is for the exclusive use of a requester.

  - File-level, block-level, and record-level locking automatically prevents two requests from accessing the same file, the same block, or the same data record simultaneously.

  - The immediate write back option causes all file modifications (delete, insert, update) to be written back to the file immediately.

  - Accidental key modification for primary keys is prevented to help ensure that your index matches the corresponding data.

- Distribution packaging - The Indexed Access Method is distributed with the following variations available:

  - A full function package that is intended to be totally resident.

  - A full function package which uses an overlay structure.

  - A totally resident package without data paging.

  - A package without data paging which uses an overlay structure.

DEVICES SUPPORTED

The Indexed Access Method supports indexed files on the following direct access devices:

- 4962 Disk Storage Unit
- 4963 Disk Subsystem
- 4964 Diskette Unit
- 4965 Diskette Unit
- 4966 Diskette Magazine Unit

In addition, the Indexed Access Method supports IBM 4969 Magnetic Tape Unit for loading and unloading indexed files with the $IAMUT1 utility.

Input control and output display devices supported for the $IAMUT1, $ILOG, and $VERIFY utilities are:

- IBM 3101 Display Station
- IBM 4978 Display Station
- IBM 4979 Display Station
- IBM 4973, 4974, and 4975 Printers
- Teletypewriter Model 33/35 or ASCII equivalent

LANGUAGES COMPATIBLE WITH INDEXED ACCESS METHOD

The following programming languages can be used to code Indexed Access Method programs.

- COBOL
- EDL
- PL/I

COMPONENTS OF INDEXED ACCESS METHOD

The Indexed Access Method consists of the following components:

- Four load modules from which you can select to support your application program Indexed Access Method requests. These load modules are named:

  - $IAM (full function with overlay)
  - $IAMRS (full function resident)
  - $IAMNP (overlay without data paging)
  - $IAMRSNP (resident without paging)

  The module you select will be named $IAM after installation.

- A load module ,$IAMSTGM, which is used to obtain the data paging area, if the data paging feature is requested.

- A set of object modules that you may use to generate a customized load module. If you use one of the four supplied load modules, you do not need the object modules.

- The object module, IAM, is called a link module. You include IAM with your application program using the linkage editor to provide the interface to the Indexed Access Method.

- Two copy code modules for inclusion in EDL programs, IAMEQU and FCBEQU. IAMEQU provides symbolic parameter values for constructing CALL parameter lists. FCBEQU provides a map of the file control blocks.

- Load modules for each of the Indexed Access Method utilities $IAMUT1, $VERIFY, and $ILOG.

CHAPTER 2. USING THE INDEXED ACCESS METHOD

The purpose of this chapter is to familiarize you with some fundamentals of the Indexed Access Method. Some of the features mentioned in the previous chapter will be described only in part here so that a basic example can be constructed. The purpose of this example is to demonstrate the ease with which you can establish an indexed file and to help you select which parts of the book apply directly to your application requirements.

YOUR DATA RECORD

The data records you wish to process with the Indexed Access Method have the following specific requirements:

- the records must contain a common field that can be used as a primary key
- each record must have a unique primary key
- the initial records to be loaded must be in ascending order by the primary key
- all records that make up an indexed file must be of the same length

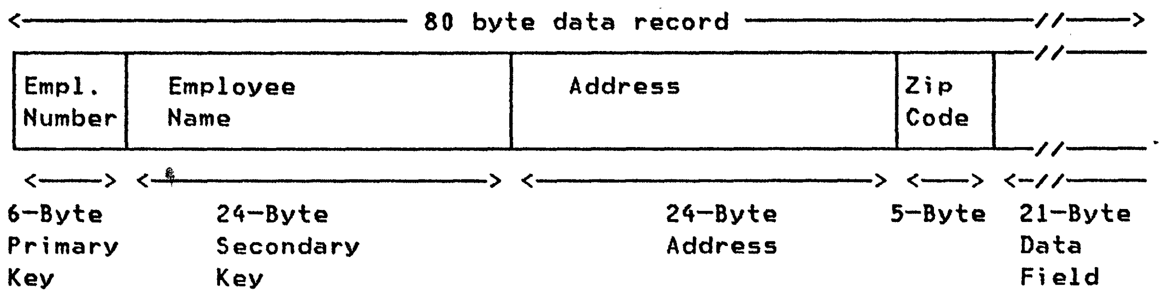
The primary key is any field you designate within your data records. The key field must begin at the same location in each record. Each key field must have the same length. The key in each record must be unique within the file (data set).

The data records that you will initially load must be in ascending order, based upon the field you use as the key. If your data records are not ready to be loaded when you define your primary indexed file, the records can be loaded later by an application program or with the LO (load) command of $IAMUT1.

Your application might use an employee number as the primary key in an indexed file for some applications. You might want to define secondary keys, such as employee name, for the same file for other applications. Using secondary keys requires a secondary index to be defined. Defining a secondary index and using secondary keys is described in "Chapter 5. Building a Secondary Index."

Whether you use the $IAMUT1 utility to load your data records into an indexed file from a sequential file, or load them with an application program, you must know the format of your input data record.

Following is a sample record layout. Although the primary key is shown starting in position 1, it could have been anywhere in the record.



The records used for our example have the following attributes:

- block size 256 bytes
- record size 80 bytes
- primary key length 6 bytes
- key position 1

SETTING UP AN INDEXED FILE USING $IAMUT1

Use the Indexed Access Method utility program, $IAMUT1, to set up an indexed file. After this utility is loaded into the system for execution, the utility displays a sequence of prompts. The prompts are questions displayed on a terminal one at a time to which you can reply using the terminal keyboard. Responding to the questions causes the utility to perform the required steps to:

1. set up the structure of the file (space for records to be loaded, free space for inserts, and an index)

2. allocate a data set (the utility prompts you for a data set and volume name and calls $DISKUT3 to allocate space for the indexed file)

3. define and format the indexed file

4. load the data records into the indexed file

Loading and using the SE (set parameter) command of the $IAMUT1 utility is described here for the purpose of our example, however, for a complete description of $IAMUT1 see "Chapter 8. The $IAMUT1 Utility."

The responses for our example are shown in **bold face type** inside the box. The bold bracketed numbers at the left, outside the box, identify explanatory remarks that we have written below the box using the same bracketed numbers. Of course these brackets and explanations do not appear on the screen when $IAMUT1 is being used.

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt is displayed as follows:

```
[1] ENTER COMMAND (?): SE
```

[1] Entering the letters **SE** (set parameters), followed by pressing the ENTER key, causes four options to be displayed:

```
    SET FILE DEFINITION PARAMETERS
    0 = EXIT
[2] 1 = SIGNIFICANT PARAMETERS
    2 = ALL PARAMETERS
    3 = PARAMETERS FROM EXISTING INDEXED DATASET
    ENTER OPTION: 1
```

[2] The response digit '1', causes prompts to follow which allows you to define an indexed file with a minimum of information. This response cause a one line prompt to be displayed.

Note: Although the following prompts are displayed one line at a time when using the utility, the prompts and responses are listed here in logical groups for simplicity in describing them.

```
[3]     SECONDARY INDEX  (Y/N):? N
                                                          DEFAULT  NEW VALUE
[4]     RECORD SIZE                                             0:80
[5]     KEY SIZE                                                0:6
[6]     KEY POSITION                                            1:1
[7]     BLOCKING FACTOR (RECORDS PER BLOCK)                     1:3
[8]     NUMBER OF BASE RECORDS                                  0:5
[9]     ESTIMATED TOTAL RECORDS                                 6:20
[10]    TYPE OF INSERT ACTIVITY(C=CLUSTERED,R=RANDOM)           C:R
[11]    DATA SET SIZE IN EDX RECORDS:                   15
[12]    INDEXED ACCESS METHOD RETURN CODE:              -1
[13]    SYSTEM RETURN CODE:                             -1
[14]    CREATE/DEFINE FILE (Y/N)?: Y
```

[3] The first prompt, "SECONDARY INDEX (Y/N)?:" asks if you are specifying a secondary index. The response was **N** for no, because we are defining the parameters for a primary indexed file.

[4] The second prompt, "RECORD SIZE" requests the length that the records are to be in the indexed file which you are defining.

Note that there are two columns near the right-hand edge of the display. The column on the left is headed by the word "DEFAULT". In the default column the values are listed that will be used in setting up the file if no value is supplied in the response (only the ENTER key is pressed). The column on the right, headed "NEW VALUE" is where the decimal value is placed from your keyboard response, followed by pressing the ENTER key.

In this example we are using a record length of **80**.

[5] The "KEY SIZE" prompt is for the length of the primary key in the data record. In this example we are using a key which is **6** bytes long.

[6] Our key field begins in position **1** of the data record.

[7] We are requesting that our indexed file be blocked with **3** records in each 256-byte block.

[8] The number of base record slots to be defined is **5**. This number is based on the number of data records we plan to load. You cannot **load** more records than this value, however, it does not restrict you from **inserting** new data records in the **free** (empty) slots later.

[9] The total number of records that we anticipate that this resultant indexed file will ever contain is **20**.

[10] The type of record insert activity is to be R (random). The records added to this file will be inserted by an application program when those records are available.

The choice of random or clustered is based on the type of record additions that are anticipated. Random is chosen when the records to be added are expected to be evenly distributed throughout the file.

Clustered is chosen when the records to be added are expected to be in groups, relative to their range in key value.

[11] Following the previous response the system will display the number of records required to contain an indexed file using the parameters you have supplied.

[12] The Indexed Access Method return code (-1) indicates that the parameters you supplied are acceptable; no Indexed Access Method rules have been violated.

[13] The system return code (-1) should always be -1 if the Indexed Access Method return code is -1. If any errors are encountered, the return code may provide additional information.

[14] If you have verified that the parameters you entered are correct, the data set (file) size in EDX records is acceptable, and the return codes are both -1, you can reply Y and the file will be defined and created.

If you wish to change any of the parameter values that you previously supplied, respond N to this prompt and you will be prompted for the next command. To re-enter your responses, reply SE and the prompt sequence will be repeated.

A Y in response to this prompt causes the next prompt sequence to begin.

```
[15]  ENTER DATA SET (NAME,VOLUME):  IAMFILE,EDX003
      NEW DATA SET IS ALLOCATED
[16]  DO YOU WANT IMMEDIATE WRITE-BACK?  Y .
[17]  INVOKE LOAD/REORGANIZE AFTER CURRENT FUNCTION (Y/N)? Y
      DEFINE IN PROGRESS
      DATA SET SIZE IN EDX RECORDS:             15
      INDEXED ACCESS METHOD RETURN CODE:        -1
      SYSTEM RETURN CODE:                       -1
[18]  INVOKE LOAD (L), REORGANIZE (R) OR END (E) ?L
      LOAD ACTIVE
```

[15] The data set and volume name you reply to this prompt is what $DISKUT3 uses to allocate a data set for your file. A successful allocation results in the information message "NEW DATA SET IS ALLOCATED".

[16] The immediate write back option is recommended for most applications. It means that we want any record in the indexed file that we process with our application program to be written back to the indexed file immediately. Otherwise, the record will be held in a buffer until that buffer is needed by the Indexed Access Method.

[17] If you have data records in a sequential data set which you want $IAMUT1 to load for you at this time, reply Y to this prompt.

If you are going to load the data records with your application program, reply N.

For our example we responded Y which causes the following information messages to be displayed: "DEFINE IN PROGRESS". The information messages then inform you with the data set (file) size in EDX records. The system return code and Indexed Access Method return code values at the completion of the file allocation and formatting are displayed for your information.

[18] Because of the Y response to the previous prompt, this prompt is generated to allow you to:

- load base records (as shown in this example)
- reorganize an existing indexed file for loading into the file being defined
- end the current SE command session

We are going to load records during this session so the response is Y. Following the 'LOAD ACTIVE' information message, the prompts continue.

```
[19] $FSEDIT FILE RECSIZE = 128
[20] INPUT RECORD ASSUMED TO BE     80 BYTES. OK?: N
[21] ENTER RECORD SIZE: 128
[22] ENTER INPUT  BLOCKSIZE (NULL = UNBLOCKED): 256
[23] ENTER INPUT  DATA SET (NAME,VOLUME): SEQ01,EDX003
     INPUT REC GT OUTPUT REC. TRUNCATION WILL OCCUR  •
[24] OK TO PROCEED:? Y
     LOAD IN PROCESS
     END OF INPUT DATA SET
[25] ANY MORE DATA TO BE LOADED?: N
        5 RECORDS LOADED
     LOAD SUCCESSFUL
```

The next sequence of prompts refers to the input data set containing the data records that are going to be loaded into the indexed file.

[19] The utility accepts input records which have been prepared by the Event Driven Executive utility $FSEDIT. The $FSEDIT record size is specified as 128.

[20] Because the output data set (indexed file) records are 80 bytes, this prompt determines whether the input sequential data set is also an 80 byte record data set.

If you use the Event Driven Executive edit utilities to prepare your data records for input to the Indexed Access Method, remember that these utilities place one 80-byte line from $FSEDIT in a 128-byte record. The first record begins at location 1, and the second record begins at location 129. Two of these 128-byte records make one 256-byte EDX record.

Because we used $FSEDIT, we responded **N**.

[21] This prompt requests the input data record attributes.

Because our input data records were created by $FSEDIT, our 80-byte records were converted to 128-byte records. Therefore, our response is **128**.

[22] The Indexed Access Method utility, $IAMUT1 accepts your records as either unblocked (one record per block) or blocked (more than one record per block) input. The utility prompts you for the block size of the input data set being loaded. If the input data set is unblocked, reply to the block size prompt by pressing the Enter Key. See "Blocked and Unblocked Sequential Data Sets" on page IG-164 for a description of blocked and unblocked sequential data sets.

If your input data records are unblocked sequential, reply by pressing the Enter Key. If your input is blocked sequential, reply with the actual blocksize that was used to prepare your input data records.

Our example uses blocked sequential records, created on every line by $FSEDIT, with a blocksize of 256.

[23] Reply to this prompt with your input data record data set and volume name. Our response was SEQ01,EDX003.

[24] This prompt verifies whether truncation of the input records is acceptable. Because our record size specified is actually 80 bytes long, but we responded 128 because $FSEDIT converts the records to 128 bytes, the following warning message is displayed. "INPUT REC GT OUTPUT REC. TRUNCATION WILL OCCUR" This means that the extra bytes attached by $FSEDIT to our 80-byte data records will now be removed. The response is Y.

The information message "LOAD IN PROCESS" tells us that $IAMUT1 is reading the input data set and loading the input data records into the base record slots. The information message "END OF INPUT DATA SET" indicates that the end-of-file condition, on the input data set, has been encountered.

[25] This prompt allows you to specify another input data set, if more data records are to be loaded from another data set. In this example, only 1 data set is being used and the response of N caused the records loaded statistics to be displayed, followed by the "LOAD SUCCESSFUL" message.

The design of an indexed file varies according to your application. A comprehensive approach to designing your indexed files begins with "Defining The File Structure With $IAMUT1" on page IG-19.

PROCESSING THE INDEXED FILE

Now that the indexed file has been defined, formatted, and loaded with data records, the file is ready for an application program to access any of the records in the indexed file for processing. An application program might use the following EDL coded requests to open the indexed file and retrieve a record.

```
     *
     *  OPEN THE INDEXED FILE FOR PROCESSING
     *
                  .
                  .
                  .
[1]             CALL   IAM,(PROCESS),IACB,(DS1),(OPENTAB),(SHARE)
     *
     *  PERFORM A DIRECT RETRIEVAL OF THE RECORD WHOSE KEY IS JONES PW
     *
[2]             CALL   IAM,(GET),IACB,(BUFF),(KEY1)
                  .
                  .
                  .
     KEY1       TEXT   'JONES PW'
     OPENTAB    DATA   F'0'
                DATA   A(IAMERR)
                DATA   F'0'
     IACB       DATA   F'0'
```

[1] This Indexed Access Method request opens the primary index file in process mode so that other requests can be issued for processing records in the indexed file.

[2] This Indexed Access Method request retrieves a record from the indexed file. The primary key of this record contains the name 'JONES PW'.

Functions of the Requests

Following is a list of functions that you can perform using the Indexed Access Method requests in your application program:

Initiate general purpose access to an indexed file with a PROCESS request. After the PROCESS request has been issued, any of the following functions can be requested:

- Direct reading - Retrieving a single record independently of any previous request.

- Direct updating - Retrieving a single record for update; complete the update by either replacing or deleting the record.

- Sequential reading - Retrieving the next logical record relative to the previous sequential request.

  The first sequential request can access the first record in the file or any other record in the file.

- Sequential updating - Retrieving the next logical record for update; complete the update by either replacing or deleting the record.

- Inserting - Placing a single record, in its logical key sequence, into the indexed file.

- Deleting - Removing a single record from the indexed file.

- Extracting - Extracting data that describes the file.

Note that the update functions require more than one request.

When a function is complete, another function may be requested, except that a sequential processing function can be followed only by another sequential function. You can terminate sequential processing at any time by issuing a DISCONN or ENDSEQ request. An end-of-data condition also terminates sequential processing.

A complete list of the Indexed Access Method requests, the operand descriptions, and correct syntax is described in "Chapter 7. Coding the Indexed Access Method Requests" on page IG-97. There are also coding examples using the Indexed Access Method requests in three programming languages in "Appendix C. Coding Examples." The languages used in the examples are Event Driven Language, COBOL, and PL/I. The purpose of these examples is not to show any particular application, but to help you when planning and writing your application program.

SUMMARY

This chapter has introduced some fundamentals of using the Indexed Access Method. The references in this chapter to other chapters in this manual were placed there to help you select the specific information you need for your application. A list of those references is repeated here to assist you in locating the detailed information on the listed subjects.

- For a complete description of $IAMUT1 see "Chapter 8. The $IAMUT1 Utility"

- A comprehensive approach to designing your indexed files is described in "Chapter 3. Defining Primary Index Files"

- Defining a secondary index for using secondary keys is described in "Chapter 5. Building a Secondary Index"

- Description of blocked and unblocked sequential data sets is described in "Blocked and Unblocked Sequential Data Sets" on page IG-164

- The complete list of Indexed Access Method requests, the operand descriptions, and correct syntax is described in "Chapter 7. Coding the Indexed Access Method Requests"

- Guide line information on processing the indexed file is located in "Chapter 7. Coding the Indexed Access Method Requests." This guideline information should be read prior to planning and coding your application program.

CHAPTER 3. DEFINING PRIMARY INDEX FILES

This chapter presents the following major topics:

- Primary Indexed Files
- Designing Indexed Files Using $IAMUT1 - option 1
- Designing Indexed Files Using $IAMUT1 - option 2
  - Indexed Access Method Blocks
  - Index Blocks
  - File Control Block
  - File Structure Types
  - Option 2 Examples
- Designing Indexed Files Using $IAMUT1 - option 3
- Defining/Creating, and Loading A File - Summary

This chapter provides information for defining indexed files and is arranged according to your option selection when using $IAMUT1. The beginning of the chapter has information which applies to any type of primary index file design. That general information section is followed immediately with an example using $IAMUT1, option 1. The option 2 section is next and contains information that you will need to know prior to designing an index file with $IAMUT1, option 2. The fourth section applies to using $IAMUT1, option 3.

PRIMARY INDEX FILES

A primary index file contains data records, a multilevel index, control information, and it can optionally contain free space.

Free space can be distributed throughout the file and at the end of the file. Free space provides areas for inserting new records and is described later.

In an indexed file, the records are arranged in ascending order by key.

DATA RECORD PRIMARY KEY

The primary key can be any field within your data record that you select, however, it must meet the following requirements:

- the selected field must start at the same location in each record
- all portions of the key field must be contiguous
- the primary key length cannot exceed 254 bytes
- the field must contain data that is unique within the data set

Defining the Key

Define a single key field by specifying its size and position in the record when you select the file formatting parameters using the SE (set parameter) command of the $IAMUT1 utility. The longer the key, the larger the index. The key should not be longer than necessary but long enough to ensure uniqueness. A shorter key is more efficient than a long key.

ENSURING UNIQUENESS OF THE KEY: To identify each record in an indexed file, each primary key must be unique. If key duplication is possible, the key field must be expanded to ensure that it is unique.

For example, customer name is a key which may involve duplicates. To avoid duplication, lengthen the key field to include other characters such as part of the customer address or the account number. Because the characters in the key must be contiguous, you may need to rearrange the fields in the record.

Another way to eliminate duplication is for you to modify new records dynamically whenever a duplication occurs during loading or processing. One or more characters at the end of the key field can be reserved for a suffix code. Whenever a duplicate occurs, add a value to the suffix and make another attempt to add the record to the file. The result is a file that can contain a sequence of keys such as SMITH, SMITH1, and SMITH2. If you add a suffix, you must use the entire unique key when accessing a record directly.

Providing Access by More Than One Key

To provide good performance with both direct and sequential access, each indexed file is indexed by a single primary key. At times, however, it may be useful to locate records by a secondary key. For example, in a customer file indexed by account number, you might want to locate a record by customer name.

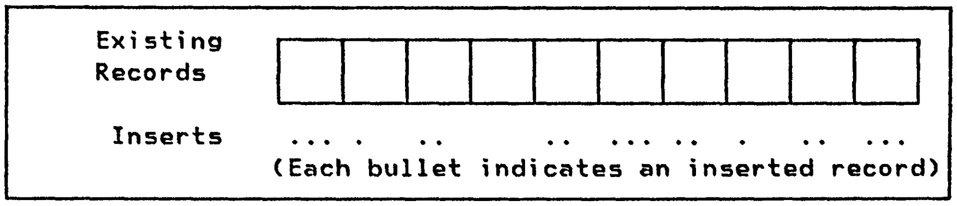
To provide access by a secondary key, you must build a secondary index (a separate file). For a description of setting up secondary indexes, see "Chapter 5. Building a Secondary Index" on page IG-67.

RANDOM AND CLUSTERED RECORD INSERTS

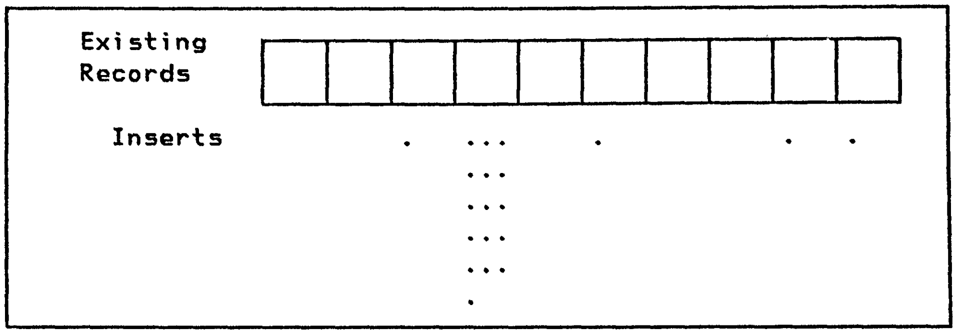
The Indexed Access Method permits records to be added to an existing file. The records are inserted by the Indexed Access Method in the proper locations according to their key value. This keeps the keys throughout the indexed file in ascending sequence.

Records to be inserted are sometimes required to be distributed throughout the file rather evenly, other times the records to be inserted are in groups.

When there are more individual records to be inserted throughout the file, based on their key value, than there are groups of records to be inserted, this is called <u>random</u> record inserts. The following diagram represents random inserted records among existing records.



Record inserts are considered <u>clustered</u> if most of the inserts occur at only certain places in the file. The following diagram represents clustered inserts by vertically stacked bullets.



DEFINING THE FILE STRUCTURE WITH $IAMUT1

Defining an indexed file structure is the process of analyzing the file requirements and selecting the appropriate file parameters. This allows you to either precisely define your indexed file or, by proper option selection, $IAMUT1 will define most of the parameters for you.

$IAMUT1 is a <u>prompt driven</u> utility. When it is loaded, messages are displayed requesting information to be entered on a keyboard. The responses you enter through the keyboard determine how the utility will operate.

The SE command of the $IAMUT1 utility permits you to select one of three options for defining your indexed file. The parameter selections are made using the SE command of the $IAMUT1 utility. The SE (set parameters) command of $IAMUT1 provides three options for you to choose from to define your indexed file as follows:

1. Option 1 significant parameters - allows you to define an indexed file by supplying a minimum of information. The description of your data records is required and whether you expect **random** or **clustered** record insert activity.

2. Option 2 all parameters - allows more flexibility in precisely defining your indexed file but requires more parameters to be supplied.

3. Option 3 parameters from existing indexed data set - can be used when you have an existing indexed file and you wish to use the same parameters for a new indexed file.

$IAMUT1 Option Selection Guide

Having read the preceding material, you are probably ready to make a choice as to which option you want to use in defining your indexed file. The following table will help you to find the appropriate information, based on your indexed file defining objectives.

Your Objective	Option	Information location
You want the Indexed Access Method to calculate and structure your file	1	See "Designing Indexed Files Using $IAMUT1 - Option 1" on page IG-21
You want to structure a file and provide specific information for the parameters	2	See "Designing Indexed Files Using $IAMUT1 - Option 2" on page IG-24
You want the Indexed Access Method to structure a file using the parameters of an existing file	3	See "Designing Indexed Files Using $IAMUT1 - Option 3" on page IG-57

DESIGNING INDEXED FILES USING $IAMUT1 - OPTION 1

Option 1 is used if you need to set up your indexed file quickly and easily. You specify only the necessary information and the utility determines the proper values for other parameters. An indexed file generated with this option may not be optimum in terms of storage space performance.

If you want to supply more parameters than are available with this option, or you wish to set up a totally dynamic indexed file, you should see "Designing Indexed Files Using $IAMUT1 - Option 2" on page IG-24. If you already have an indexed file established and you wish to use those same parameters, you should see "Designing Indexed Files Using $IAMUT1 - Option 3" on page IG-57.

OPTION 1

The Indexed Access Method utility, $IAMUT1, option 1 of the SE (set parameters) command, provides you with the opportunity to select only those parameters necessary to set up an indexed file.

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1] ENTER COMMAND (?): SE
```

[1] Entering **SE** causes the following option list prompt to be displayed.

```
    SET FILE DEFINITION PARAMETERS
    0 = EXIT
[2] 1 = SIGNIFICANT PARAMETERS
    2 = ALL PARAMETERS
    3 = PARAMETERS FROM EXISTING INDEXED DATA SET
    ENTER OPTION: 1
```

[2] Respond to this prompt by entering the digit '1'. This response causes a one line prompt from the next prompt sequence to be displayed.

> **Note:** Although the following prompts are displayed one line at a time when using the utility, the entire prompt list is shown for simplicity in describing the parameters.

```
[3]   SECONDARY INDEX  (Y/N)?: N
                                                     DEFAULT  NEW VALUE
[4]   RECORD SIZE                                            0:80
[5]   KEY SIZE                                               0:40
[6]   KEY POSITION                                           1:1
[7]   BLOCKING FACTOR (RECORDS PER BLOCK)                    1:3
[8]   NUMBER OF BASE RECORDS                                 0:5
[9]   ESTIMATED TOTAL RECORDS                                6:20
[10]  TYPE OF INSERT ACTIVITY(C=CLUSTERED,R=RANDOM)          C:R
      DATA SET SIZE IN EDX RECORDS:                 12
      INDEXED ACCESS METHOD RETURN CODE:            -1
      SYSTEM RETURN CODE:                           -1
[11]  CREATE/DEFINE FILE  (Y/N)?:
```

[3] The first line asks, are you specifying a secondary index. The response should be **N** for no, because you are defining the parameters for a primary index file.

[4] The record length shown is 80, however, the entry you will make is the actual record length you want your indexed file records.

[5] Enter the length of your data record field that you are using as the key field. The maximum primary key length is 254.

[6] Enter the position where your primary key field begins. Your data record begins with 1.

[7] Specify the blocking factor (number of records per block) you want your indexed file to have. Remember that when a record is accessed, an entire block is actually read into the system buffer.

[8] Enter the number of base record slots to be defined. This value is the number of records you will load initially. You cannot **load** more records than this value specifies.

[9] Enter the total number of records you expect this file to contain. records you plan to insert during processing.

[10] Enter the type of record insert activity you expect to have

[11] If you have verified that the parameters you entered are correct, the data set (file) size in EDX records is acceptable, and the return codes are both -1, you can reply Y and you can create and define the file. If you wish to change any of the parameters, reply **N** and you can reenter the SE command and enter any new values for the parameters.

Replying N terminates the SE function and you can return to this point by reentering the SE command or the DF command (within the same session of $IAMUT1). The DF command of $IAMUT1 is described under "DF - Define Indexed File" on page IG-146.

To review the prompts that occur when Y is replied at this point return to the example in "Chapter 2. Using the Indexed Access Method."

DESIGNING INDEXED FILES USING $IAMUT1 - OPTION 2

Option 2 is used if you have performed an analysis of your file requirements and you want to precisely define your primary indexed file. This option provides a wide range of parameters to allow you to specify your file structure in detail. You can optimize the file structure according to your application requirements for the best storage use and performance.

If you want to supply only the minimum parameters you might want to use option 1 which is described earlier in this chapter under "Designing Indexed Files Using $IAMUT1 - Option 1" on page IG-21. If you already have an indexed file established and you wish to use those same parameters, you should see "Designing Indexed Files Using $IAMUT1 - Option 3" on page IG-57.

OPTION 2

The following information is provided so that you can supply the required information to the prompts when defining a primary index file using option 2 of $IAMUT1. The information is organized in levels of Indexed Access Method blocks. The material should be read sequentially because it provides the information which must be understood in order to apply it to the examples which are placed near the end of this option 2 material.

INDEXED ACCESS METHOD BLOCKS

Indexed files consist of three kinds of blocks:

- Data blocks, which contain records

- Index blocks, which contain pointers to data blocks or lower-level index blocks

- File control blocks, which contain control information

Following is an overview diagram showing the types of blocks and their general relationships to each other in an indexed file.

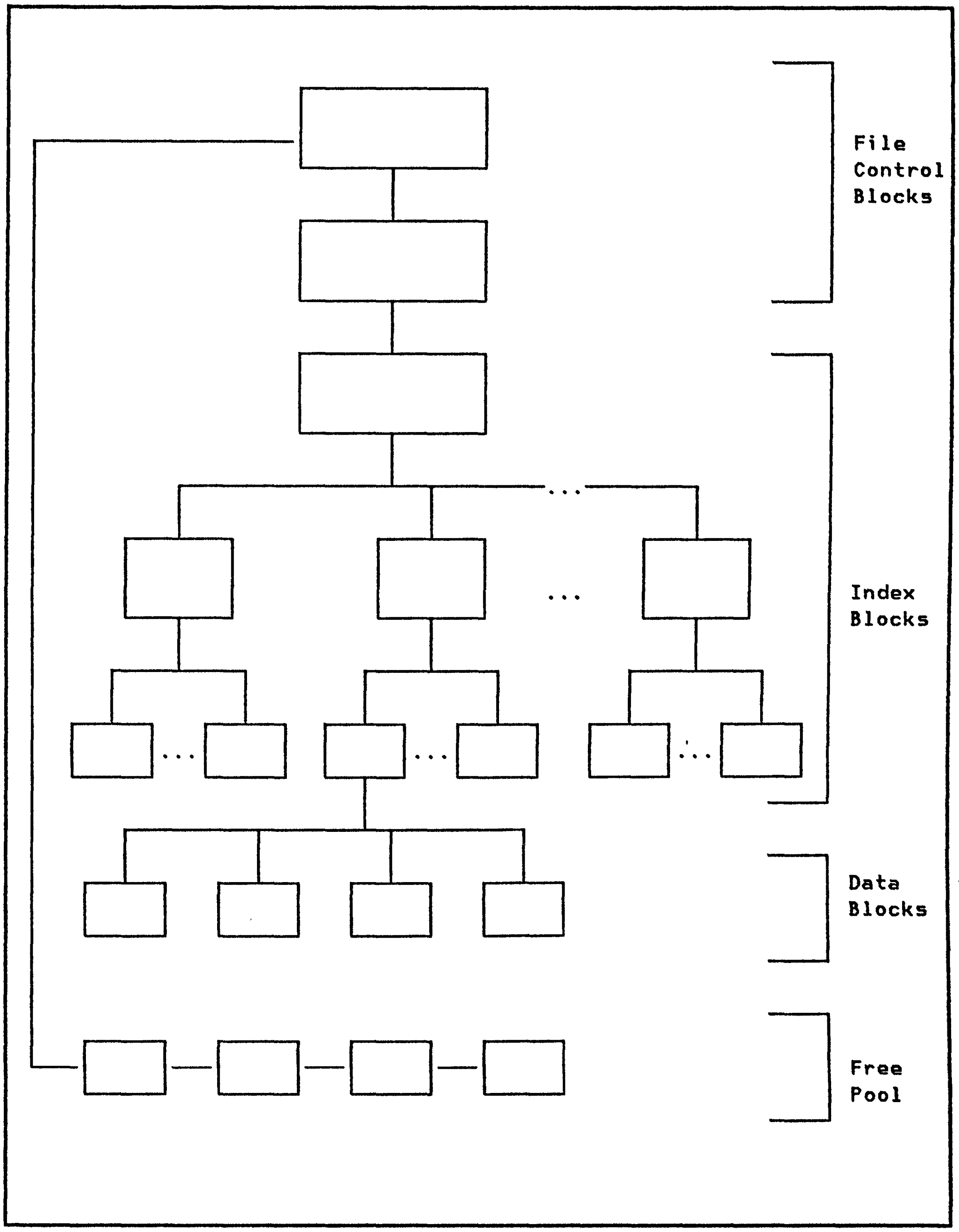


Figure 1. Indexed File Logical Structure

The indexed file is composed of a number of fixed length blocks. The block is the unit of data transferred by the Indexed Access Method between disk/diskette and the central buffer. Block size must be a multiple of

256. A block is addressed by its relative block number (RBN). The first block in the file is located at RBN 0.

Note that the RBN is used only in indexed files by the Indexed Access Method. An Indexed Access Method block differs from an Event Driven Executive record in the following ways:

1. The size of a block is not limited to 256 bytes; its length can be a multiple of 256.

2. The RBN of the first block in an indexed file is 0. The record number of the first Event Driven Executive record in a file is 1.

The size, in 256-byte records, of the file is calculated by the SE command of the $IAMUT1 utility.

As stated initially, three kinds of blocks exist in an indexed file: data blocks, index blocks, and file control blocks. These blocks are all the same length, as defined by BLKSIZE, but they contain different kinds of information. Data blocks contain data records, index blocks contain index entries, and file control blocks contain control information.

DATA BLOCKS

Each data block contains a header, one or more data records, and it can contain free space for additional data records.

The records in each data block are in ascending order, according to the key field in each record.

Each data block header contains the address of the next sequential data block, providing sequential processing capability.

A data block contains a header followed by data records. The number of records that can be contained in a data block depends on the size of the data block and the size of the record. The header of the block is 16 bytes.

The number of record areas in the block is:

$$\boxed{\frac{\text{block size - 16}}{\text{record size}}}$$

The result is truncated; any remainder represents the number of unused bytes in the block. For example, if block size is 256 and record size is 80, the data block can accommodate three records and there is no unused area. The key field of the last record slot in an index block is the high key for the data block even if the block is not full. However, if the last record of the block has been deleted, the key field of the last record slot will contain a key higher than that of any other record in the block. Deletion of a record does not reduce the key range for the block unless the block is emptied. Figure 2, shows the format of a data block.

FREE SPACE

When an indexed file is loaded with base records, free space is reserved for records that may be inserted during processing. There are four kinds of free space: free records, free blocks, reserve blocks, and reserve index entries.

FREE RECORDS: Free records are areas reserved at the end of each data block. The FREEREC parameter of the SE command of $IAMUT1, specifies the number of free records that are reserved in each data block. The remaining record areas are called allocated records.

For example, if a block contains three data record areas and you specify one free record per block, then there are two allocated records per block. For the layout of a data block containing two allocated records and one free record, see Figure 2.

When records are loaded (file is open in load mode), the allocated records are filled, and the free records are skipped. When additional records are inserted (file is open in process mode), free records are used to hold inserted records.

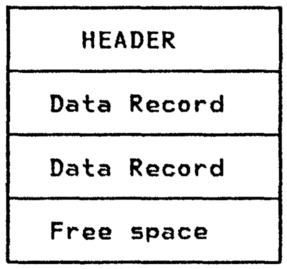


Figure 2. Data Block Format Example

For an example of specifying FREEREC, see "Example 1: Allocating Free Records" on page IG-43.

FREE BLOCKS: Free blocks follow the allocated data blocks within each cluster. Free blocks have all of their records marked as free records. The FREEBLK parameter of option 2 is used to specify the percentage of blocks that are to be marked as free blocks.

When records are loaded, the allocated record areas in the allocated data blocks are filled, and the free blocks are skipped. During processing, as data blocks become full, a free block provides space for insertions.

For an example of specifying FREEBLK, see "Example 2: Allocating Free Records and Free Blocks" on page IG-45.

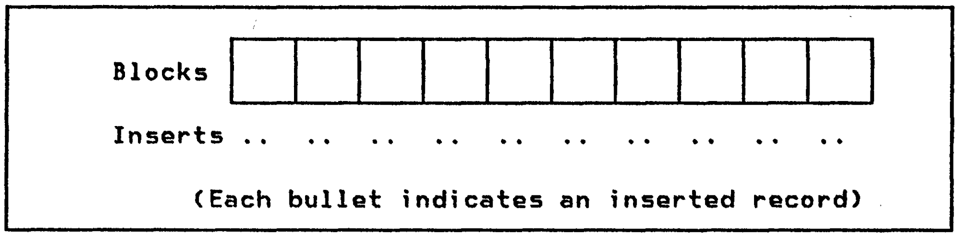
SEQUENTIAL CHAINING: Data blocks in an indexed file are chained together by forward pointers located in the headers of data blocks. Only allocated data blocks are included in the sequential chain. Chaining provides for sequential processing of the file with no need to reference the index. When a free block is converted to an allocated block, the free block is included in the chain.

Reserving Space For Record Inserts

If base records are to be loaded and record insertions are expected in random locations throughout the file, use BASEREC to reserve the number of base records. Use some combination of the following parameters: FREEREC to reserve free records in each data block, FREEBLK to reserve free blocks in each cluster (group of blocks), and DYN to provide a free pool.

For example, consider a file with 5 records per block, and 10 data blocks per cluster. Suppose that the file consists of 300 base records and 200 inserts.

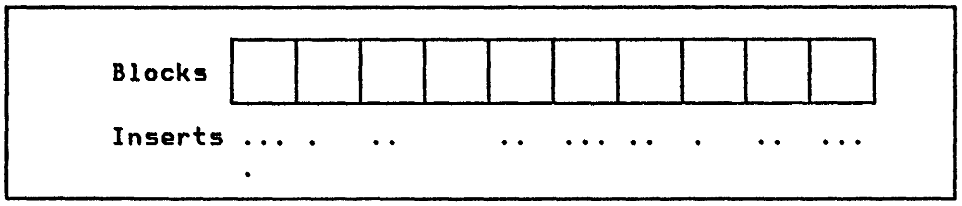
If the inserts are distributed evenly throughout the file, the pattern of inserts is:


Blocks
Inserts .. ..
(Each bullet indicates an inserted record)

With this kind of distribution you can specify 2 free records per block to absorb the inserts; no free blocks or free pool are needed.

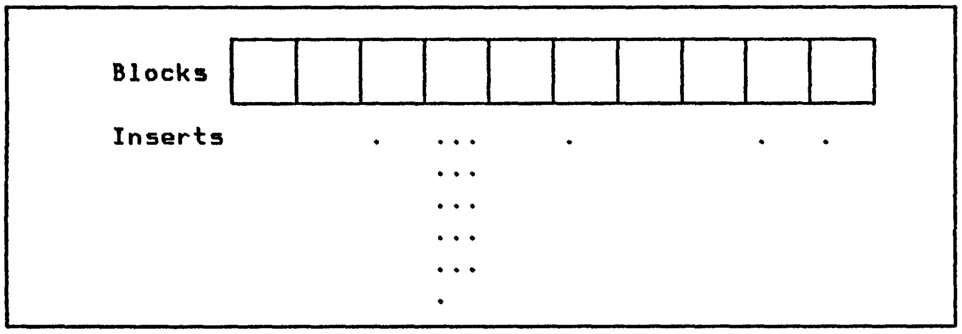
Of course inserts do not usually occur in such an even pattern. Free blocks help to absorb a concentration of inserts. The more uneven the expected distribution, the greater the free block specification should be.

Suppose the same number of inserts is distributed in this pattern:



With this distribution, specify either 3 free records per block, or 20% free blocks with 2 free records per block.

Now suppose the distribution were more uneven:



In this case a satisfactory mix of free space is 1 free record per block and 40% free blocks. An alternative is to use 1 free record per block and the DYN parameter to hold those record inserts of more than 1 record per block.

Calculating Data Blocks

This calculating information is provided for your convenience if you choose to calculate the number of blocks for a specific file. For reference later there is a summary of all calculations in "Appendix A. Summary of Calculations" on page IG-223. However, $IAMUT1 automatically calculates the required data blocks based on the parameters you provide. The utility also lists at file definition time (when using the SE command) the number of blocks required according to your parameter values.

The number of allocated data blocks in a file is the specified number of base records (BASEREC) divided by the number of allocated records per data block, with the result rounded up if there is a remainder.

For example, suppose you intend to load 1000 records in an indexed file that is formatted for two allocated records and one free record per block and five allocated blocks and one free block per cluster. The number of allocated blocks in a file is:

$$\frac{\text{number of base records}}{\text{number of allocated records per block}}$$

The number of allocated blocks in this example is 1000/2 or 500 blocks.

INDEX BLOCKS

An index block contains a header followed by a number of index entries. Each index entry consists of a key and a pointer. The key is the highest key associated with a lower level block; the pointer is the RBN of that block. The number of entries contained in each index block depends on block size and key size. The header of the block is 16 bytes. The RBN field in each entry is 4 bytes. The key field in each entry must be an even number of bytes in length; if the key field is an odd number of bytes in length, the field is padded with one byte to make it even. The number of index entries in an index block is:

$$\frac{\text{block size} - 16}{4 + \text{key length}}$$

The result is truncated; any remainder represents the number of unused bytes in the block.

For example, if block size is 256 and key length is 28, then each index entry is 32 bytes, there are 7 entries in a block, and the last 16 bytes of the block are unused.

PRIMARY INDEX BLOCKS (PIXB)

A set of data blocks is addressed (described) by a single primary index block (PIXB). Each key in the index block is the highest key in the data block that its accompanying relative block number (RBN) addresses. A block is addressed by its RBN. The PIXB and the data blocks it describes are called a cluster.

Clusters

Primary-level index blocks and data blocks are stored together in the file in groups called clusters. Each cluster consists of a primary-level index block and as many data blocks and free blocks as it points to. For example, if there are seven entries in an index block, there are eight blocks in a cluster: one primary-level index block and up to 7 data/free blocks. If reserve blocks have been specified, the blocks represented by the reserve block entries are not included until insert activity has taken place and the required blocks have been obtained from the free pool. For example, if there are seven entries in an index block and one of the entries is a reserve block entry, the cluster consists of seven blocks (one index block and six data blocks). See Figure 3 for a cluster example.

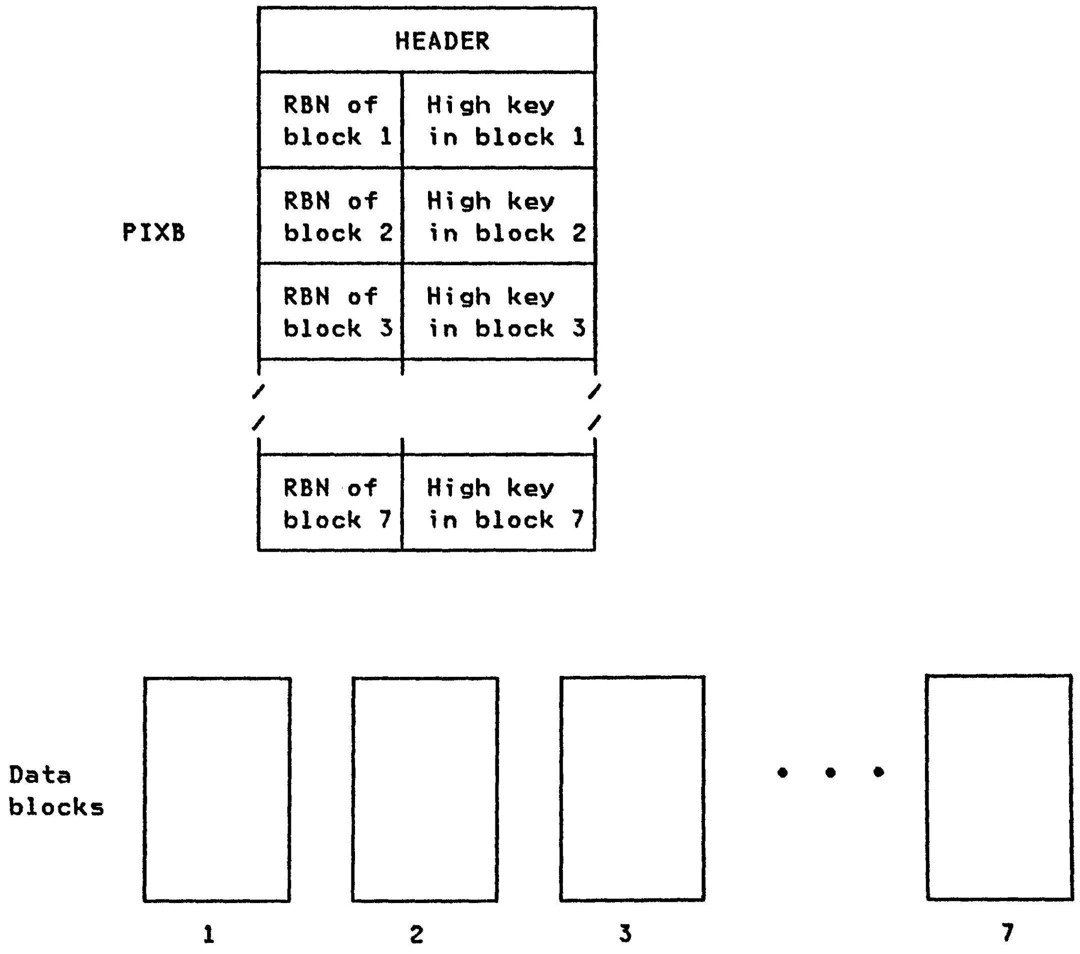


Figure 3. Cluster Example

Primary-Level Index Blocks

Entries in a primary-level index block point to data blocks. Each entry primary-level index block is one of three possible types:

- allocated entry
- free block entry
- reserve block entry

ALLOCATED ENTRY: An allocated entry points to an active data block. The key portion of the entry is initialized to binary ones by the $IAMUT1 utility. After records have been loaded or written to a data block, the key portion of the entry which points to the data block contains the highest key from the data block.

The pointer portion contains the RBN of the data block. Allocated entries are the first entries in an index block. The number of index entries allocated, when the indexed file is initially created, is the total number of entries per index block, less the number of entries of the other two types (free block entry and reserve block entry).

FREE BLOCK ENTRY: A free block entry points to a free data block. The key portion of the entry contains binary zeros. The pointer portion contains the RBN of the free block. Free block entries follow the allocated entries in the index block. The number of index entries formatted as free entries when the indexed file is initially created is the specified percentage (FREEBLK) of the total number of entries in an indexed block, with the result rounded up if there is a remainder.

RESERVE BLOCK ENTRY: A reserve block entry does not point to a block but is reserved for later use as a pointer to a data block which can be taken from the free pool. Both the key and pointer portions of a reserve block entry are binary zeros. Reserve block entries are at the end of the index block. When a reserve block entry is converted to a used entry, the index block is reformatted to move the entry to the allocated entry area of the block.

Reserve blocks do not exist in the cluster. When all data blocks in a cluster are used and another data block is needed, a data block can be created from the free pool. If the primary-level index block contains a reserve block entry, it is used to point to the record from the free pool. The reserve block entry in the primary-level index block points to the block, and the data block becomes an allocated data block.

The number of index entries initially formatted as reserve block entries is the specified percentage (RSVBLK) of the total number of entries, with the result rounded up if there is a remainder. However, if the number of free block entries plus the number of reserve block entries require all index entries, the number of reserve block entries is reduced by 1, providing at least one allocated entry per index block.

To calculate the number of primary-level index blocks in an indexed file, you must know the initial number of data blocks allocated in the indexed file.

Calculating Clusters

This calculating information is provided for your convenience if you choose to calculate the number of blocks for a specific file. However, $IAMUT1 automatically calculates the required data blocks based on the parameter values you provide. The utility also lists at file definition time (when using the SE command) the number of blocks required according to your parameter values.

The number of clusters in a file is the number of allocated data blocks divided by the number of allocated entries in each primary-level index block, with the result rounded up if there is a remainder.

$$\boxed{\frac{\text{allocated blocks}}{\text{allocated entries in each PIXB}}}$$

Note that in the calculation, if the quotient is not an integer, it is rounded up (rather than truncated) in order to accommodate all of the base records.

The number of free blocks in the file (not including the free pool) is the number of clusters in the file multiplied by the number of free entries in each primary-level index block.

The Last Cluster

The last cluster in the file may be different from the other clusters. It contains the same number of free blocks as the other clusters but only enough allocated blocks to accommodate the records that you have specified with the parameter BASEREC. Because rounding occurs in calculating the number of clusters, a few more allocated records than required may exist in the last allocated block. The last cluster can be a short one because only the required number of blocks are used.

If the number of allocated blocks divided by the number of allocated blocks per cluster leaves a remainder, the remainder represents the number of allocated entries in the primary-level index block in the last cluster. Unused entries in the last primary-level index block are treated as reserve block entries.

The initial number of data blocks is the specified number of base records (BASEREC) divided by the number of allocated records in a data block, with the result rounded up if there is a remainder.

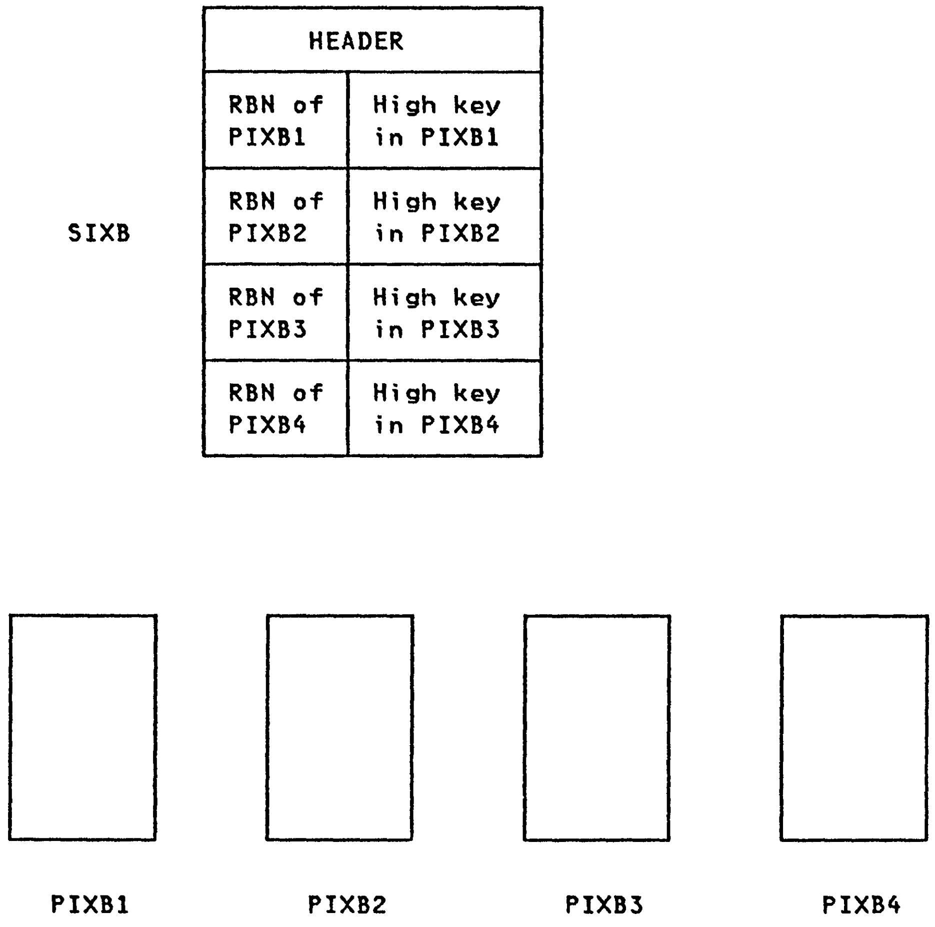
$$\boxed{\frac{\text{BASEREC}}{\text{data records per block}}}$$

The number of primary-level index blocks is the initial number of allocated data blocks divided by the number of allocated entries per primary-level index block, with the result rounded up if there is a remainder.

$$\boxed{\dfrac{\text{allocated data blocks}}{\text{allocated entries per primary-level index block}}}$$

SECOND-LEVEL INDEX BLOCKS (SIXB)

If the file is large enough to require more than one cluster, each PIXB (or cluster) has an entry in a second-level index block (SIXB). The entry in a SIXB contains the address of the PIXB and the highest key in the cluster. The SIXB has the following structure:



Entries in a second-level index block point to primary-level index blocks. Each entry in a second-level index block is one of two possible types:

- allocated entry
- reserve index entry

ALLOCATED ENTRY: An allocated entry points to an existing primary-level index block. The key portion of the entry is initialized to binary ones by the $IAMUT1 utility. After records have been loaded or written, the key portion of the entry contains the highest key from the primary-level index block. The pointer portion contains the RBN of the primary-level index block. Allocated entries are the first entries in the index block. The number of index entries allocated when the indexed file is loaded is calculated as the total number of entries per index block, less the number of reserve index entries.

RESERVE INDEX ENTRY: A reserve index entry does not point to a block but is reserved for later use as a pointer to a primary-level index block that can be taken from the free pool. Both the key and pointer portions of a reserve index entry are binary zeros.

Reserve index entries, in second-level index blocks, provide index space for the index structure to be expanded by adding new primary-level index blocks. These, in turn, can have data blocks associated with them, thus forming new clusters. This process of forming a new cluster is called a <u>cluster split</u>.

For an example of using RSVIX, refer to "Example 4: Allocating Reserved Index Entries" on page IG-51.

Reserve index entries are at the end of the index block. The number of index entries initially formatted as reserve index entries is the specified percentage (RSVIX) of the total number of entries, with the result rounded up if there is a remainder. However, if the number of reserve index entries is the same as the total number of entries in an index block, the number of reserve index entries is reduced by 1, providing at least one allocated entry per second-level index block.

The number of second-level index blocks is the number of primary-level index blocks divided by the number of allocated entries per second-level index block, with the result rounded up if there is a remainder.

$$\frac{\text{number of PIXBs}}{\text{allocated entries per SIXB}}$$

HIGHER-LEVEL INDEX BLOCK (HIXB)

If the file is large enough to require more than one SIXB, the SIXBs in the file are described by one or more higher-level index blocks (HIXB) in the same manner as the SIXB describes PIXBs. There is always one index block that describes the entire file.

The index of an indexed file is constructed in several levels so that, given a key, there is a single path (one index block per level) cascading through the index levels that leads to the data block associated with that key. The index is built from the bottom up. At the lowest level are the primary-level index blocks. At the second level are index blocks containing entries that point to the primary-level index blocks. The highest level of the index structure consists of a single index block.

Entries in a higher-level index block point to index blocks at the next lower level. All entries in higher-level index blocks are allocated entries. The key portion of the entry contains the highest key from the index block of the next lower level. The pointer portion contains the RBN of the next lower level index block. The number of blocks at any higher index level is the number of index blocks at the next lower level divided by the total number of entries per index block, with the result rounded up if there is a remainder.

If the number of index blocks at any level is one, that level is the top level of the index. Although the Indexed Access Method is capable of initially defining and supporting 17 levels of index, an indexed file is formatted with only as many index levels as are required for the number of records. If an indexed file has not been fully loaded and one or more higher index levels have not yet been required, the unnecessary higher levels are not used, even though they exist in the file structure.

INDEX EXAMPLE: Assume that 500 data blocks are allocated to a file and that each primary-level index block contains one free block entry, one reserve block entry, and five allocated entries. Therefore, the total number of primary-level index blocks is 100. Each second-level index block contains one reserve index entry and six allocated entries; therefore, the number of second-level index blocks is 17. The number of entries in higher level index blocks is seven, resulting in three index blocks at the third level and one at the fourth level.

Therefore the file contains a total of 121 index blocks of which 100 are primary-level index blocks, 17 are second-level index blocks, 3 are third-level index blocks, and 1 is a fourth-level index block. This distinction is important because high-level index blocks are located contiguously at the beginning of the file (after the FCB), while primary-level index blocks are scattered throughout the file with the data blocks. Figure 4 on page IG-38 shows the structure of the higher-level index blocks.

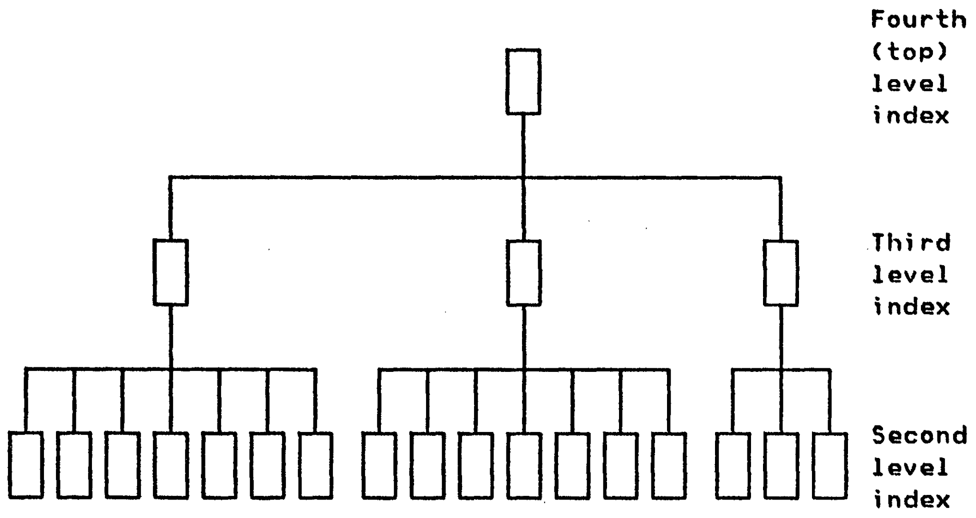


Figure 4. High-level Index Structure

FREE POOL

If you specify that you want a free pool, your indexed file contains a pool of free blocks at the end of the indexed file. The file control block contains a pointer to the first block of the free pool, and all blocks in the free pool are chained together by forward pointers.

A block can be taken from the free pool to become either a data block or an index block. The block is taken from the beginning of the chain, and its address (RBN) is placed in the appropriate primary-level index block (if the new block is to become a data block) or in the second level index block (if the new block is to become a primary-level index block), and so on. Any block in the free pool can be used as either a data block or as an index block.

When a data block becomes empty because of record deletions, the data block may return to the free pool (depending on the delete threshold (DELTHR) parameter). If the data block is returned to the free pool, reference to the block is removed from the primary-level index block, and the block is placed at the beginning of the free pool chain.

Calculating the initial size of the free pool consists of the following steps:

- Each reserve block entry in a primary-level index block represents a potential data block from the free pool. The number of data blocks that can be assigned to initial clusters is the number of primary-level index blocks times the number of reserve block entries in each primary-level index block.

- Each reserve index entry in a second-level index block represents a potential primary-level index block from the free pool. The number of primary-level index blocks that can be assigned from the free pool into the index structure set up at file definition time is the number of second-level index blocks multiplied by the number of reserve index entries in each second-level index block.

- Each primary-level index block taken from the free pool consists entirely of empty (reserve block) entries. New data blocks can be taken from the free pool for the entries in the new primary-level index block. The number of data blocks is the number of entries per index block multiplied by the number of new primary-level index blocks (calculated in the previous step).

- The maximum number of blocks that can be taken from the free pool and placed into the index structure set up at file definition time is the sum of the previous three calculations.

- The actual number of blocks in the free pool is determined in one of two ways:

  - The percentage (FPOOL) of the maximum possible free pool as specified by the RSVIX and RSVBLK parameters. The result is rounded up if there is a remainder. If the DYN parameter is also used, its value is added to the sum.

- The DYN parameter, if specified with no other free space parameters, allocates a free pool of the specified number of blocks.

DELTHR - DELETE THRESHOLD: The percentage (0-99) of blocks to retain in a cluster as records are deleted and blocks made available. This is known as the delete threshold **DELTHR**. When a block becomes empty, this parameter, if supplied, determines if the block should be returned to the free pool.

FILE CONTROL BLOCK

The file control block (FCB) is the first block in the file (RBN 0); it contains control information.

Indexed files have an FCB Extension as the second block. The FCB Extension contains the parameters used to define the file.

> **Note:** Indexed files built with a version of the Indexed Access Method prior to version 2 do not contain an FCB extension.

You can access the FCB and FCB Extension by either of the following methods:

- using the EXTRACT function in an EDL program
- using the $VERIFY utility

You can locate the field names in the FCB and FCB Extension by examining a listing of FCBEQU, a copy code module that is supplied as part of the Indexed Access Method. The FCB Extension contains the parameters that were used to set up the file using the $IAMUT1 SE command. Control information is also contained in block headers; a description of control information is contained in "FCB Extension Report" on page IG-196.

FILE STRUCTURE TYPES

A wide range of file structure is available. You can set up files that vary from the totally _dynamic_ to the highly _structured_. Whether a file is structured or dynamic depends on the degree to which it uses a _free pool_.

A free pool is an area in your indexed file which contains a pool of free blocks. The file control block contains a pointer to the first block of the free pool, and all blocks in the free pool are chained together by forward pointers. A block can be taken from the free pool to become either a data block or an index block.

Dynamic files offer the advantage of easy file design and good space utilization. They have the disadvantage of a potential performance decrease.

Structured files offer the advantage of good performance. They have the disadvantage of a more complex file design and greater space requirements.

Either method can result in a need to reorganize the file; the structured approach because the file can run out of space for inserts, and the dynamic approach because of performance considerations.

The type of indexed file to be defined, structured or dynamic, therefore, depends on the file requirements and the efficiency required.

Structured File

A structured file has its base record slots, free space, and the index structure needed to support them built at file definition time by the Indexed Access Method utility using the file structure parameters you specify. The structured file uses little, if any, free pool. The structured file offers better performance than the dynamic file but can result in unused space.

Whether or not a structured file has a free pool depends on whether or not you supply a value for the DYN parameter when the file is defined. When the DYN parameter is used, the FREEREC, FREEBLK, RSVBLK, RSVIX, and FPOOL parameters, if supplied, are also used in establishing the structured free space. The number and types of blocks in a structured file are the result of calculated values you supply as parameters when defining the file. Most of the blocks are not taken dynamically from the free pool as they are needed because they are established at file definition time.

Dynamic File

The higher the degree to which a file uses a free pool, the more dynamic it is; the system builds index and data blocks for you as they are needed.

The Indexed Access Method provides a dynamic file restructuring capability. It makes use of any free pool space the file has, even if the file is mostly structured.

The Indexed Access Method can restructure a file in two ways:

- As records are inserted and additional space is needed in specific areas of the file, blocks are taken from the free pool and become data blocks where needed. If additional index blocks are needed, blocks are taken from the free pool for this purpose as well. Index blocks can be added at any level, and the number of levels of index can increase as needed. This function is performed automatically by the Indexed Access Method on any file that has a free pool associated with it.

- As records are deleted and blocks become empty, they are returned to the free pool. If index blocks become empty (because the blocks under them have been returned to the free pool) they are also returned to the free pool. This helps to maintain a supply of blocks in the free pool to be used if other areas of the file expand.

For an example of defining a totally dynamic file, see "Example 5 - Defining a Totally Dynamic File" on page IG-54.

USING THE DYN PARAMETER: The DYN parameter can be used to adjust how much the free pool is used. This adjustment varies how dynamic a structured file is.

In a totally dynamic file, the initial file defined consists of only the file control blocks, one primary index block and one data block. The rest of the file is in the free pool.

To define a totally dynamic file, you need to only supply a value for the DYN parameter to allow the rest of the file to be assigned to the free pool.

A dynamic file can be used when the records you want to add to the file are not sorted into ascending key sequence. In that case, you can place the records in the file by inserting them in random sequence. The Indexed Access Method will place them in their proper sequence within the indexed file.

If base records are to be loaded initially and they are sorted in ascending key sequence but insert activity is unknown, you can use a totally dynamic file design. Use the BASEREC parameter to reserve the number of base record slots required. Use the DYN parameter to provide the free pool needed for record inserts.

Note: When a dynamic file has grown to its working size, it should be reorganized for more efficient operation.

OPTION 2 EXAMPLES

The examples which follow are provided to show the option 2 prompts and the effects of certain parameter values. Although the values used are small for simplicity of explanation, they are usually much larger in an actual application. Also a given example does not represent a complete primary index file but addresses a particular part of a file and its associated parameters which we wish to describe at that place in the chapter.

EXAMPLE 1: ALLOCATING FREE RECORDS

The indexed file created using these parameters has only one type of free space, called free records:

```
     ENTER COMMAND (?): SE

     SET FILE DEFINITION PARAMETERS
     0 = EXIT
     1 = SIGNIFICANT PARAMETERS
     2 = ALL PARAMETERS
     3 = PARAMETERS FROM EXISTING INDEXED DATASET
     ENTER OPTION: 2
     SECONDARY INDEX (Y/N)?: N
     PARAMETER  DEFAULT NEW VALUE
     BASEREC       NULL:10
     BLKSIZE          0:256
     RECSIZE          0:80
     KEYSIZE          0:40
     KEYPOS           1:1
[1]  FREEREC          0:1
     FREEBLK          0:0
     RSVBLK        NULL:
     RSVIX            0:
     FPOOL         NULL:
     DELTHR        NULL:
     DYN           NULL:
     TOTAL LOGICAL RECORDS/DATA BLOCK:                  3
     FULL RECORDS/DATA BLOCK:                           2
     INITIAL ALLOCATED DATA BLOCKS:                     5
     INDEX ENTRY SIZE:                                 44
     TOTAL ENTRIES/INDEX BLOCK:                         5
     FREE ENTRIES/PIXB:                                 0
     RESERVE ENTRIES/PIXB(BLOCKS):                      0
     FULL ENTRIES/PIXB:                                 5
     RESERVE ENTRIES/SIXB:                              0
     FULL ENTRIES/SIXB:                                 5
     DELETE THRESHOLD ENTRIES:                          5
     FREE POOL SIZE IN BLOCKS:                          0
     # OF INDEX BLOCKS AT LEVEL   1:                    1

     DATA SET SIZE IN EDX RECORDS:                      8
     INDEXED ACCESS METHOD RETURN CODE:                -1
     SYSTEM RETURN CODE:                               -1

     CREATE/DEFINE FILE (Y/N)?: N

     ENTER COMMAND (?):
```

[1] Because record size was specified as 80 and block size was specified as 256, there are (256-16)/80 = 3 records per block. Because FREEREC was specified as 1, there are 2 full (base) records per block and 1 free record per block. Because BASEREC was specified as 10, there are 10/(2 base records per block) or 5 initial allocated data blocks (blocks that contain base records). Because FREEBLK, RSVBLK, RSVIX, FPOOL, and DYN were not specified, there are no free blocks or free pool blocks allocated. One primary index block is needed.

The number of free blocks is calculated as follows: Free entries per PIXB times the number of index blocks at level 1.

The total blocks allocated for this file is:

Initial allocated data blocks	5
Free blocks	0
Free pool blocks	0
Index blocks	1
File control block	+ 2
	8 Total

Figure 5 illustrates the format of the indexed file that would result from these SE command parameters.

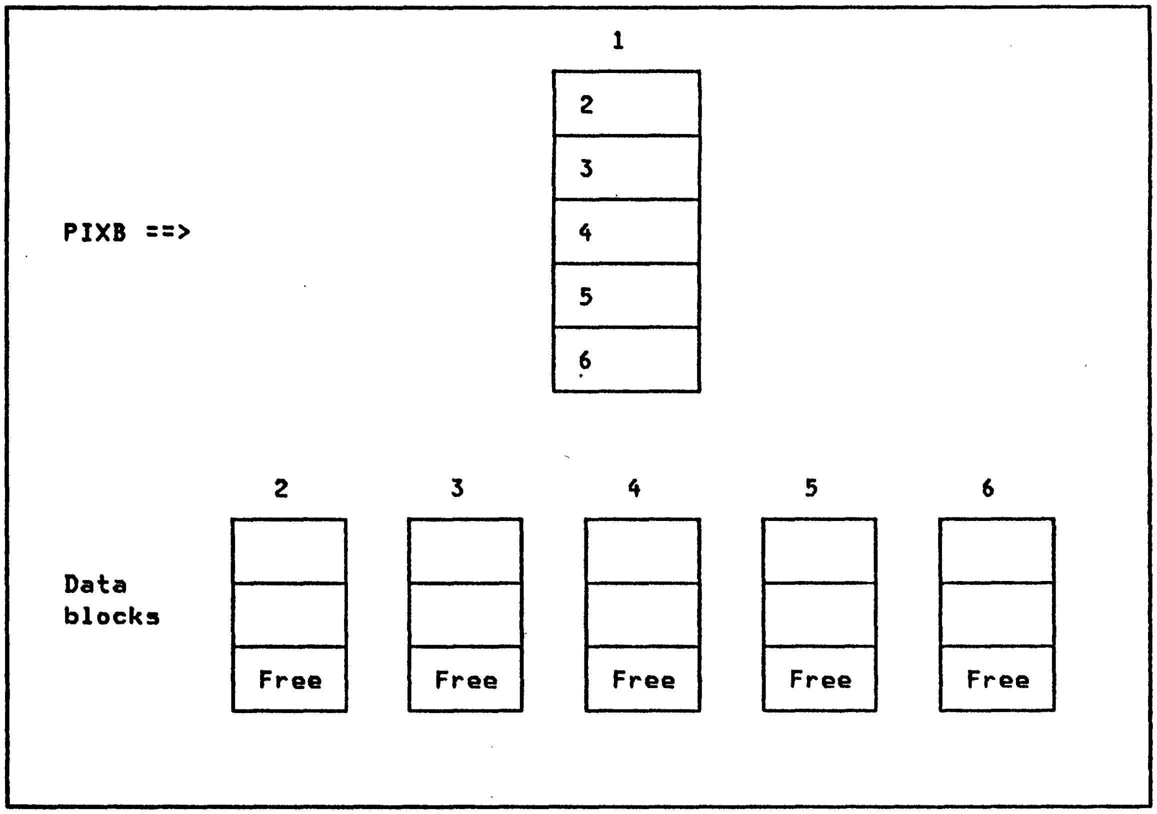


Figure 5. Indexed File with Free Records

EXAMPLE 2: ALLOCATING FREE RECORDS AND FREE BLOCKS

These parameter specifications will generate an indexed file with two types of free space — free records and free blocks:

```
     ENTER COMMAND (?): SE

     SET FILE DEFINITION PARAMETERS
     0 = EXIT
     1 = SIGNIFICANT PARAMETERS
     2 = ALL PARAMETERS
     3 = PARAMETERS FROM EXISTING INDEXED DATASET
     ENTER OPTION: 2
     SECONDARY INDEX (Y/N)?: N
     PARAMETER  DEFAULT NEW VALUE
     BASEREC       NULL:10
     BLKSIZE          0:256
     RECSIZE          0:80
     KEYSIZE          0:40
     KEYPOS           1:1
     FREEREC          0:1
[1]  FREEBLK          0:10
     RSVBLK        NULL:
     RSVIX            0:
     FPOOL         NULL:
     DELTHR        NULL:
     DYN           NULL:
     TOTAL LOGICAL RECORDS/DATA BLOCK:                   3
     FULL RECORDS/DATA BLOCK:                            2
     INITIAL ALLOCATED DATA BLOCKS:                      5
     INDEX ENTRY SIZE:                                  44
     TOTAL ENTRIES/INDEX BLOCK:                          5
     FREE ENTRIES/PIXB:                                  1
     RESERVE ENTRIES/PIXB(BLOCKS):                       0
     FULL ENTRIES/PIXB:                                  4
     RESERVE ENTRIES/SIXB:                               0
     FULL ENTRIES/SIXB:                                  5
     DELETE THRESHOLD ENTRIES:                           5
     FREE POOL SIZE IN BLOCKS:                           0
     # OF INDEX BLOCKS AT LEVEL   1:                     2
     # OF INDEX BLOCKS AT LEVEL   2:                     1

     DATA SET SIZE IN EDX RECORDS:                      12
     INDEXED ACCESS METHOD RETURN CODE:                 -1
     SYSTEM RETURN CODE:                                -1

     CREATE/DEFINE FILE (Y/N)?: N
     ENTER COMMAND (?):
```

[1] The **FREEBLK** parameter of 10 causes 10% of the total entries in each index block to point to free blocks. Because KEYSIZE was specified as 40,

the index entry size = 40 + 4 (RBN pointer) and the total entries per index block is (256-16)/44 = 5. Thus, 10% of this total rounded up is the number of free entries/PIXB (1). Because there are 5 initial allocated data blocks, one free entry and only 5 total entries per index block, 2 primary index blocks are needed. This causes a second-level index block to be allocated.

The total blocks allocated:

Initial allocated data blocks	5
Free blocks	2
Free pool blocks	0
Index blocks	3
File control block	+ 2
	12 Total

Figure 6 on page IG-47 illustrates the format of the indexed file that would result from these SE command parameters.

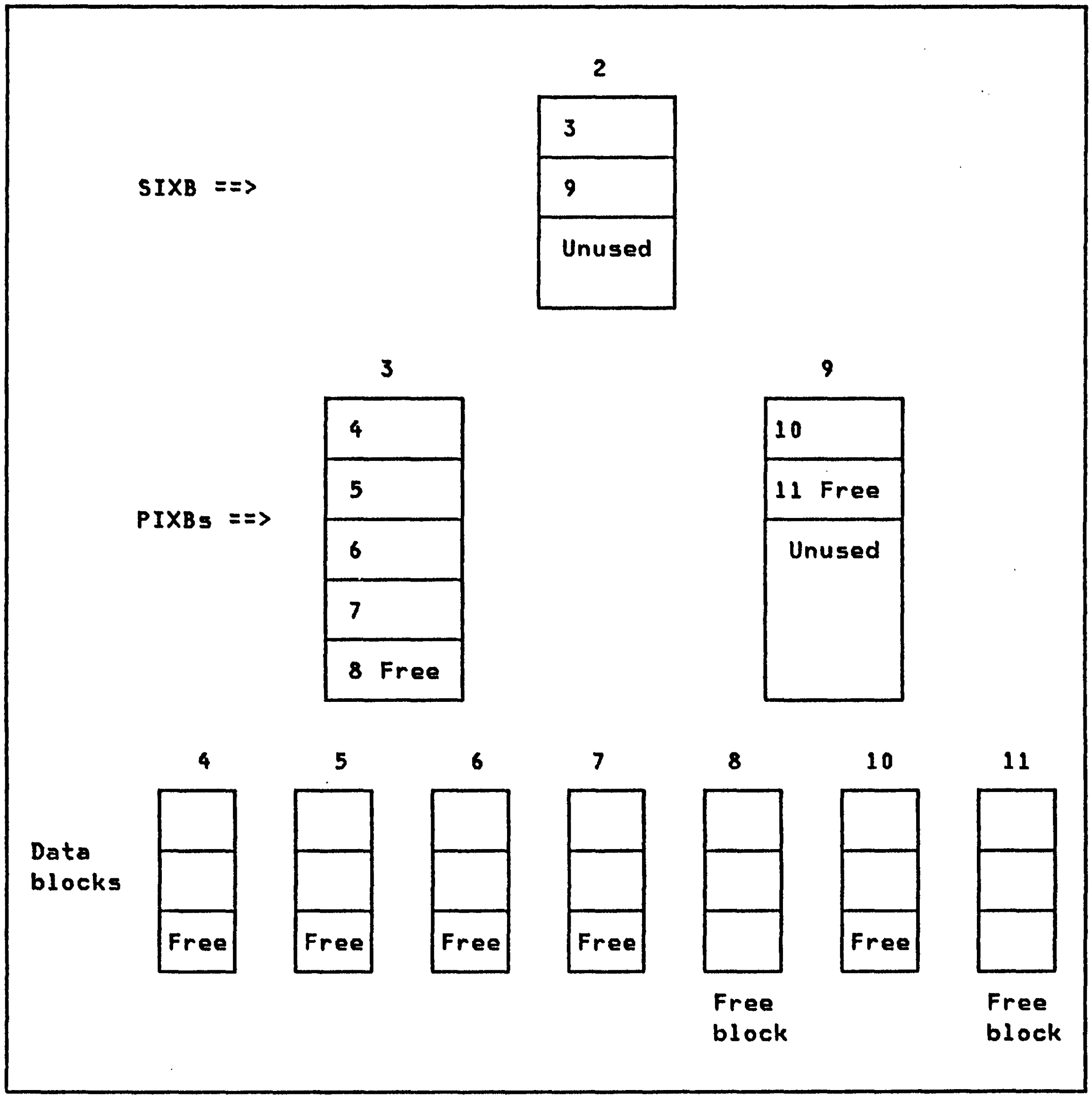


Figure 6. Indexed File with Free Records/Blocks

EXAMPLE 3: ALLOCATING RESERVED DATA BLOCKS

Reserve blocks are allocated using the RSVBLK and FPOOL parameters of the SE command. The following SE command example show the specification of an indexed file with reserved data blocks.

```
     ENTER COMMAND (?): SE

     SET FILE DEFINITION PARAMETERS
     0 = EXIT
     1 = SIGNIFICANT PARAMETERS
     2 = ALL PARAMETERS
     3 = PARAMETERS FROM EXISTING INDEXED DATASET
     SECONDARY INDEX (Y/N)?: N
     ENTER OPTION: 2
     PARAMETER  DEFAULT NEW VALUE
     BASEREC       NULL:10
     BLKSIZE          0:256
     RECSIZE          0:80
     KEYSIZE          0:40
     KEYPOS           1:1
     FREEREC          0:1
     FREEBLK          0:10
[1]  RSVBLK        NULL:10
     RSVIX            0:
[2]  FPOOL         NULL:50
     DELTHR        NULL:
     DYN           NULL:
     TOTAL LOGICAL RECORDS/DATA BLOCK:                     3
     FULL RECORDS/DATA BLOCK:                              2
     INITIAL ALLOCATED DATA BLOCKS:                        5
     INDEX ENTRY SIZE:                                    44
     TOTAL ENTRIES/INDEX BLOCK:                            5
     FREE ENTRIES/PIXB:                                    1
     RESERVE ENTRIES/PIXB(BLOCKS):                         1
     FULL ENTRIES/PIXB:                                    3
     RESERVE ENTRIES/SIXB:                                 0
     FULL ENTRIES/SIXB:                                    5
     DELETE THRESHOLD ENTRIES:                             4
     FREE POOL SIZE IN BLOCKS:                             1
     # OF INDEX BLOCKS AT LEVEL   1:                       2
     # OF INDEX BLOCKS AT LEVEL   2:                       1

     DATA SET SIZE IN EDX RECORDS:                        13
     INDEXED ACCESS METHOD RETURN CODE:                   -1
     SYSTEM RETURN CODE:                                  -1

     CREATE/DEFINE FILE (Y/N)?: N
     ENTER COMMAND (?):
```

[1] In this example **RSVBLK** was specified as 10. Thus 10% of the total entries in each PIXB will initially be reserved.

[2] Because the total entries per PIXB is 5, 10% of 5 rounded up will cause 1 entry in each PIXB to be reserved. Because there are 2 PIXBs, each with 1 reserve entry, a maximum of 2 free pool blocks can be used. However, since **FPOOL** was specified as 50%, only half of these blocks (1 block) will be allocated for the free pool.

The total blocks allocated for this file is:

Initial allocated data blocks	5
Free blocks	2
Free pool blocks	1
Index blocks	3
File control block	+ 2
	13 Total

Figure 7 on page IG-50 illustrates the format of the indexed file that would result from these SE command parameters.

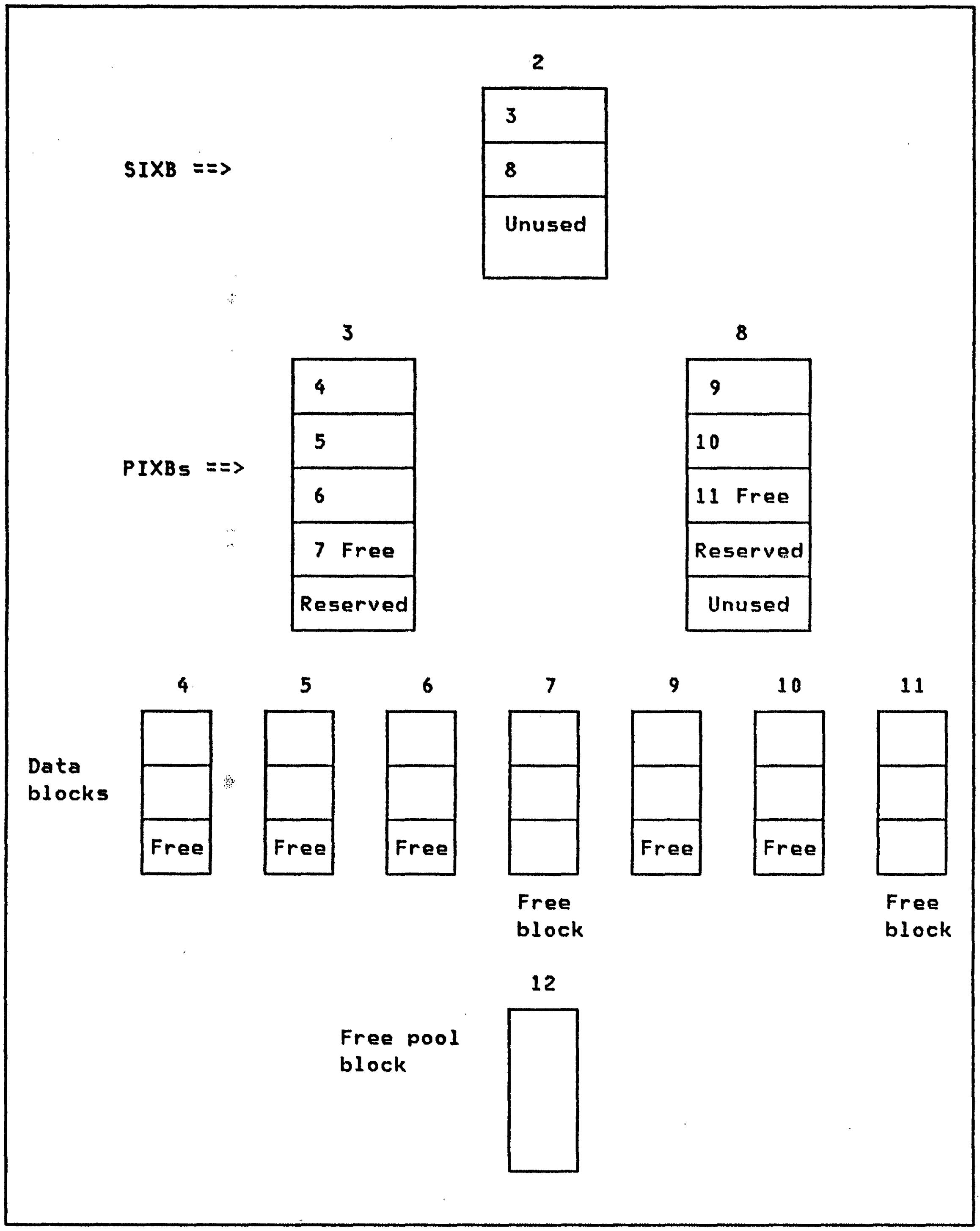


Figure 7. Indexed File with Reserved Data Blocks

EXAMPLE 4: ALLOCATING RESERVED INDEX ENTRIES

In the following example, the index structure is set up to use free pool blocks for index blocks by allocating reserve index entries using the RSVIX parameter.

```
     ENTER COMMAND (?): SE

     SET FILE DEFINITION PARAMETERS
     0 = EXIT
     1 = SIGNIFICANT PARAMETERS
     2 = ALL PARAMETERS
     3 = PARAMETERS FROM EXISTING INDEXED DATASET
     ENTER OPTION: 2
     SECONDARY INDEX (Y/N)?: N
     PARAMETER  DEFAULT NEW VALUE
     BASEREC       NULL:10
     BLKSIZE          0:256
     RECSIZE          0:80
     KEYSIZE          0:40
     KEYPOS           1:1
     FREEREC          0:1
     FREEBLK          0:10
     RSVBLK        NULL:10
[1]  RSVIX            0:10
     FPOOL         NULL:50
[2]  DELTHR        NULL:40
     DYN           NULL:25
     TOTAL LOGICAL RECORDS/DATA BLOCK:                  3
     FULL RECORDS/DATA BLOCK:                           2
     INITIAL ALLOCATED DATA BLOCKS:                     5
     INDEX ENTRY SIZE:                                 44
     TOTAL ENTRIES/INDEX BLOCK:                         5
     FREE ENTRIES/PIXB:                                 1
     RESERVE ENTRIES/PIXB(BLOCKS):                      1
     FULL ENTRIES/PIXB:                                 3
     RESERVE ENTRIES/SIXB:                              1
     FULL ENTRIES/SIXB:                                 4
     DELETE THRESHOLD ENTRIES:                          2
     FREE POOL SIZE IN BLOCKS:                         29
     # OF INDEX BLOCKS AT LEVEL   1:                    2
     # OF INDEX BLOCKS AT LEVEL   2:                    1

     DATA SET SIZE IN EDX RECORDS:                     41
     INDEXED ACCESS METHOD RETURN CODE:                -1
     SYSTEM RETURN CODE:                               -1

     CREATE/DEFINE FILE (Y/N)?: N
     ENTER COMMAND (?):
```

[1] In this example there are still 5 total entries per index block. The 10 RSVIX parameter causes 10% X 5 (rounded up to 1) of the <u>second-level</u> index block (SIXB) entries to be reserved.

In this case, 1 reserve entry is allocated in the SIXB leaving 4 full entries. Because the block pointed to by a SIXB is also an index block (PIXB), blocks in the free pool are allocated for the PIXB and the total number of data blocks it can point to. Thus the total free pool size for these parameters is 1 (reserve entry) + 5 (total entries/PIXB) + 2 (reserve block entries) = 8. Because only 50% of the total possible free pool was requested, 4 of the total free pool blocks plus the 25 blocks specified on the DYN parameter for a total of 29 blocks would be allocated to the free pool.

The total blocks allocated for this file is:

Initial allocated data blocks	5
Free blocks	2
Free pool blocks	29
Index blocks	3
File control block	+ <u>2</u>
	41 Total

[2] The percentage (0-99) of blocks to retain in the cluster as records are deleted and blocks made available. This is known as the delete threshold DELTHR. When a block becomes empty, it is first determined if the block should be given up to the free pool by checking the response to this prompt. If the block is not given up to the free pool, it is retained in the cluster, either as a free block or as an active empty block. The result of this calculation is rounded up so that any non-zero specification indicates at least one block. The calculation is adjusted to ensure that the cluster always contains at least one block. In this example, the delete threshold was specified as 40%. This results in at least 2 blocks always being retained in each cluster.

If the DELTHR parameter is specified as null (&) and DYN is not specified, DELTHR defaults to the number of allocated blocks in the cluster plus one half of the value calculated by the FREEBLK prompt.

If the DELTHR parameter is specified as null and a value is specified for the DYN parameter, DELTHR defaults to zero.

Figure 8 on page IG-53 illustrates the format of the indexed file that would result from these SE command parameters.

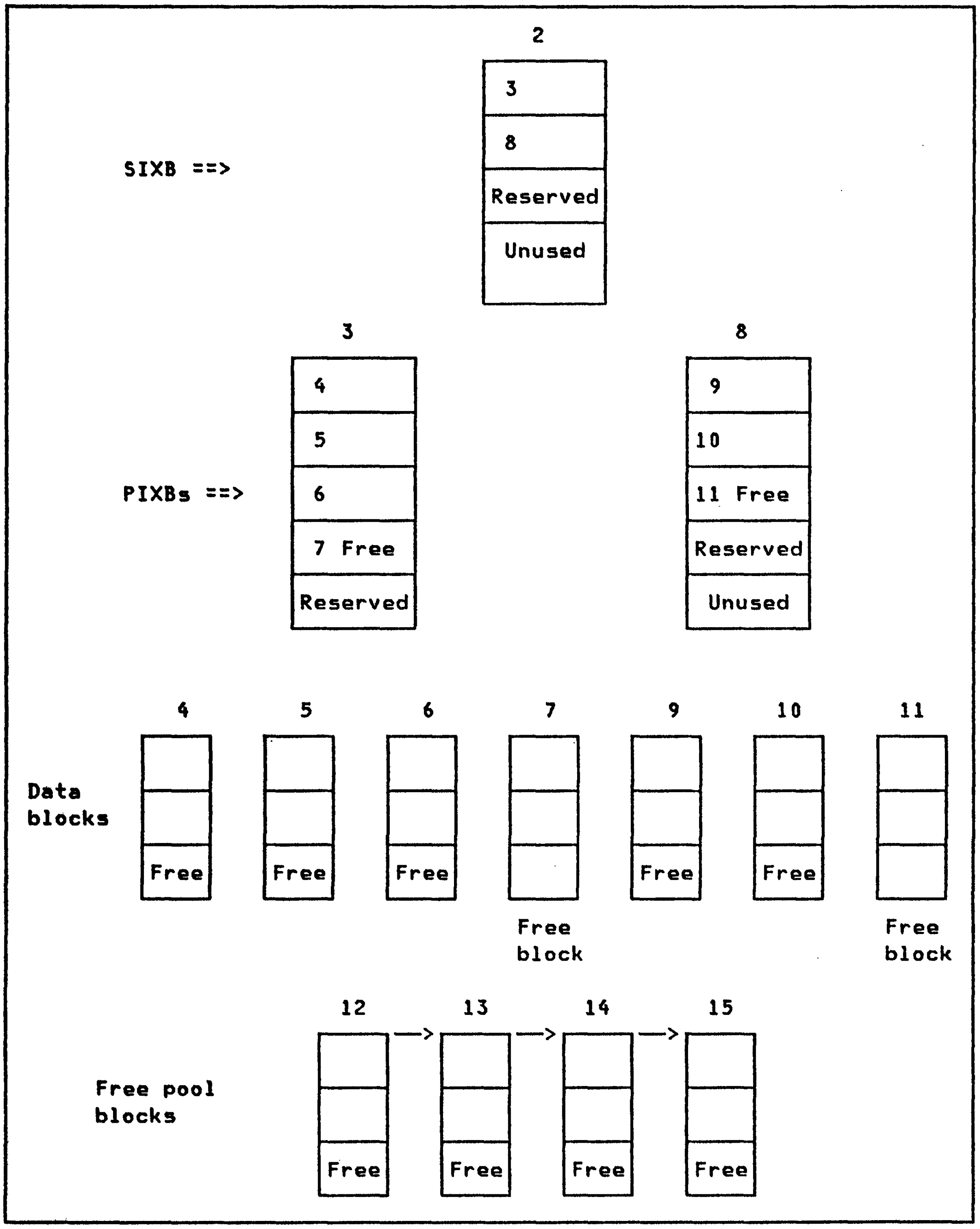


Figure 8. Indexed File with Reserved Index Entries

EXAMPLE 5 - DEFINING A TOTALLY DYNAMIC FILE

To define a totally dynamic file you need only supply the parameters which describe the format of your records within blocks: BLKSIZE, RECSIZE, KEYSIZE. If the your keys do not begin in position 1 of your records, the KEYPOS parameter must be supplied. The DYN parameter must then be specified in the number of blocks to assign to the free pool.

The following display shows the use of the SE commands of the $IAMUT1 utility to define a totally dynamic indexed file. Note that the resulting file has only one allocated data block and one index block. The rest of the space is in the free pool as specified by the DYN parameter.

```
ENTER COMMAND (?): SE

SET FILE DEFINITION PARAMETERS
0 = EXIT
1 = SIGNIFICANT PARAMETERS
2 = ALL PARAMETERS
3 = PARAMETERS FROM EXISTING INDEXED DATASET
ENTER OPTION: 2
SECONDARY INDEX (Y/N)?: N
PARAMETER  DEFAULT NEW VALUE
BASEREC       NULL:
BLKSIZE          0:256
RECSIZE          0:70
KEYSIZE          0:40
KEYPOS           1:
FREEREC          0:
FREEBLK          0:
RSVBLK        NULL:
RSVIX            0:
FPOOL         NULL:
DELTHR        NULL:
DYN           NULL:5300
TOTAL LOGICAL RECORDS/BLOCK:                3
FULL RECORDS/DATA BLOCK:                    3
INITIAL ALLOCATED DATA BLOCKS:              1
INDEX ENTRY SIZE:                          14
TOTAL ENTRIES/INDEX BLOCK:                 17
FREE ENTRIES/PIXB:                          0
RESERVE ENTRIES/PIXB (BLOCKS):              0
FULL ENTRIES/PIXB:                         17
RESERVE ENTRIES/SIXB:                       0
FULL ENTRIES/SIXB:                         17
DELETE THRESHOLD ENTRIES:                   0
FREE POOL SIZE IN BLOCKS:                5300
# OF INDEX BLOCKS AT LEVEL   1:             1

DATA SET SIZE IN EDX RECORDS:            5304
INDEXED ACCESS METHOD RETURN CODE:         -1
SYSTEM RETURN CODE:                        -1

CREATE/DEFINE FILE (Y/N)?: N

ENTER COMMAND (?):
```

The total blocks allocated for this file is:

Initial allocated data blocks	1
Free blocks	0
Free pool blocks	5300
Index blocks	1
File control block	2
	5304 Total

Figure 9 illustrates the format of the indexed file that would result from these SE command parameters.

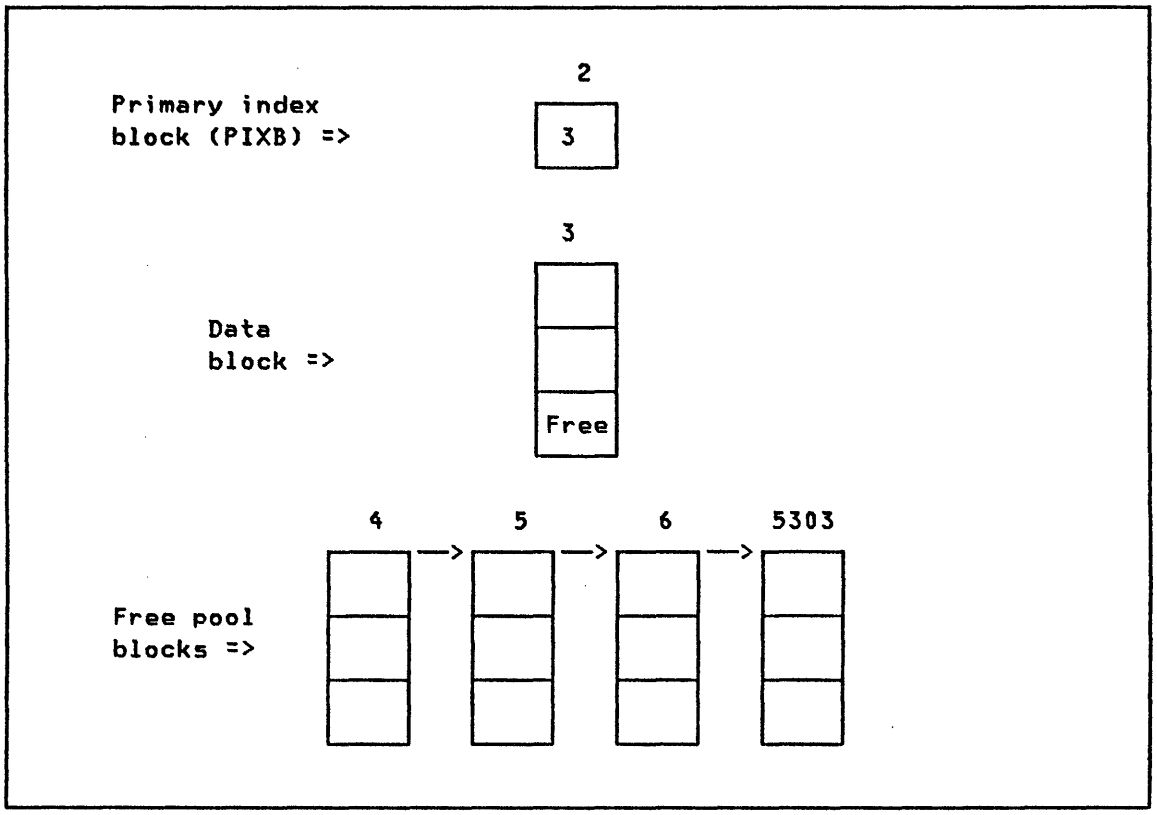


Figure 9. Totally Dynamic Indexed File

DESIGNING INDEXED FILES USING $IAMUT1 - OPTION 3

Option 3 allows you to define a new file, using the same parameters that were used to create an existing file. Using this option you are not required to manually enter any parameters. You are prompted for the data set name and volume of the existing indexed file followed by the prompt "NEW PARAMATERS EXACTLY SAME AS ORIGINAL PARAMETERS (Y/N) ?". The effects of these two possibilities are described below:

Y The new file to be defined is to appear exactly like the existing file when it was created. In other words, the parameters to be used for defining the new file will be exactly like those of the existing file.

An example of this situation is where you are satisfied with the structure of a currently existing file and now you want to build a similar file and you expect the same type of insert/delete activity.

N The growth of the existing file is to be taken into account in defining the new file. If the total number of records in the existing file do not exceed the number of base records when the file was defined, the existing file parameters will be used without change to define the new file. However, if the number of records in the existing file exceed the number of base records, the parameters for the new file will be adjusted as follows:

- BASEREC will be set as the current number of records in the existing file.
- FPOOL will be set to null.
- DYN will be set to the current number of free pool blocks in the existing file.
- All other parameters will be the same as the corresponding existing file parameters.

  Replying **N** to the prompt "NEW PARAMETERS EXACTLY SAME AS ORIGINAL PARAMETERS (Y/N)?", causes the file size to be adjusted to allow at least as many records to be loaded in the new file as appear in the existing file. This reduces the free pool amount based upon free pool depletion in the existing file.

An example of this situation is where you wish to reorganize a file. The new file should be able to handle as many records as exist in the old file.

Note: The parameters for a primary file must be set from another primary file and parameters for a secondary file must be set from another secondary file.

$IAMUT1 - OPTION 3

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1]  ENTER COMMAND (?): SE
```

[1] Entering **SE** causes the next prompt to be displayed.

```
     SET FILE DEFINITION PARAMETERS
     0 = EXIT
     1 = SIGNIFICANT PARAMETERS
     2 = ALL PARAMETERS
[2]  3 = PARAMETERS FROM EXISTING INDEXED DATA SET
     ENTER OPTION: 3
```

[2] Respond to this prompt by entering the digit '3'. This response causes the following prompts to be displayed.

```
     SECONDARY INDEX (Y/N)?: N
[3]  NAME OF EXISTING INDEXED DATASET (NAME,VOLUME): EMPLFILE,EDX003
[4]  NEW PARAMETERS EXACTLY SAME AS ORIGINAL PARAMETERS (Y/N)? Y
     DATA SET SIZE IN EDX RECORDS:                 15
     INDEXED ACCESS METHOD RETURN CODE:            -1
     SYSTEM RETURN CODE:                           -1
[5]  CREATE/DEFINE FILE (Y/N)?: N
```

[3] Enter the name of the data set and volume whose values you wish this data set to copy.

[4] If all of the parameter values used to define the existing file initially are satisfactory, reply Y. However, if you want to change any of the parameters, based on current file status, or you want to reorganize the existing file, reply N. Replying N will cause the parameter values for BASEREC and FPOOL to be adjusted so that you can load as many records into the new file as are now contained in the existing file.

[5] If you have verified that the parameters you entered are correct, the data set (file) size in EDX records is acceptable, and the return codes are both -1, you can reply Y and the file can be defined and created. If

you wish to change any of the parameters, reply N and you can reenter the SE command and enter any new values for the parameters.

DEFINING, CREATING, AND LOADING A FILE - SUMMARY

This chapter has presented the structure, content and principles of primary index files. Several examples have been used to show what results given parameter values when defining a primary index file.

In those examples the use of the SE command of $IAMUT1 was used extensively. In replying to the SE prompt "DEFINE/CREATE FILE (Y/N)?:", N was used in this chapter. This allows you to reenter the SE command and go through the prompt sequence again, changing any parameter values as required.

To see the result of replying Y, see the example used in "Chapter 2. Using the Indexed Access Method" on page IG-5. When you reply Y to the DEFINE/CREATE prompt, you enter the function called defining the file. You can enter the define file directly anytime that $IAMUT1 is loaded by responding with **DF** to the prompt "ENTER COMMAND (?):." Using the DF (define file) command is described in detail under "DF - Define Indexed File" on page IG-146.

When you reply Y to the prompt, "INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION ?:", you are given the opportunity to enter the $IAMUT1 functions of load, reorganize, or end. While in the SE function, load, reorganize, and end can be entered by replying with the letters L, R, or E, respectively. However, these functions can be entered directly from the prompt "ENTER COMMAND (?):" with **LO** for load, **RO** for reorganize, or **EN** for end.

Using the LO (load) command is described in detail under "LO - Load Indexed File" on page IG-162.

Using the RO (reorganize) command is described in detail under "RO - Reorganize Indexed File" on page IG-171.

Entering EN (end) terminates the current session of the SE command of $IAMUT1. Entering EN to the prompt "ENTER COMMAND (?):" will then terminate the $IAMUT1 utility.

CHAPTER 4. LOADING THE PRIMARY INDEX FILE

This section describes the process and methods of loading a file.

You can use two methods to load base records:

- The $IAMUT1 utility
- An application program

The methods are described in the following sections.

LOADING THE PRIMARY INDEX FILE

The Indexed Access Method uses two modes to place records into an indexed file:

1. Load mode: records are loaded sequentially in ascending order by key, skipping any free space. The records loaded are called **base records**. Each record loaded must have a key higher than any key already in the file.

2. Process mode: records are inserted in their proper key position relative to records already in the file. Records are inserted using the free space that was skipped during loading or, if a record has a new high key, it is placed in a base record after the last loaded record. If no base records are available, it is placed in the free space after the last loaded record.

The total number of base records that can be loaded is established when the indexed file is defined by the $IAMUT1 utility. It is not necessary, however, to load all (or any) base records before processing can begin. The file can be opened for loading some of the base records, closed and then reopened for processing (including inserts), and later opened for loading more base records. Figure 10 on page IG-62 illustrates this sequence.

> **Note:** Programs written in COBOL are an exception to this: COBOL programs can use load mode only once for any given indexed file. Therefore, all base records loaded in load mode must be loaded together. Base records loaded later must be inserted in process mode (with slower performance).

The amount of free space for inserts (if any) is specified using the $IAMUT1 utility when the indexed file is built. This free space can be distributed throughout the file in the form of free records within each data block, free blocks within each cluster, and in a free pool at the end of the file.

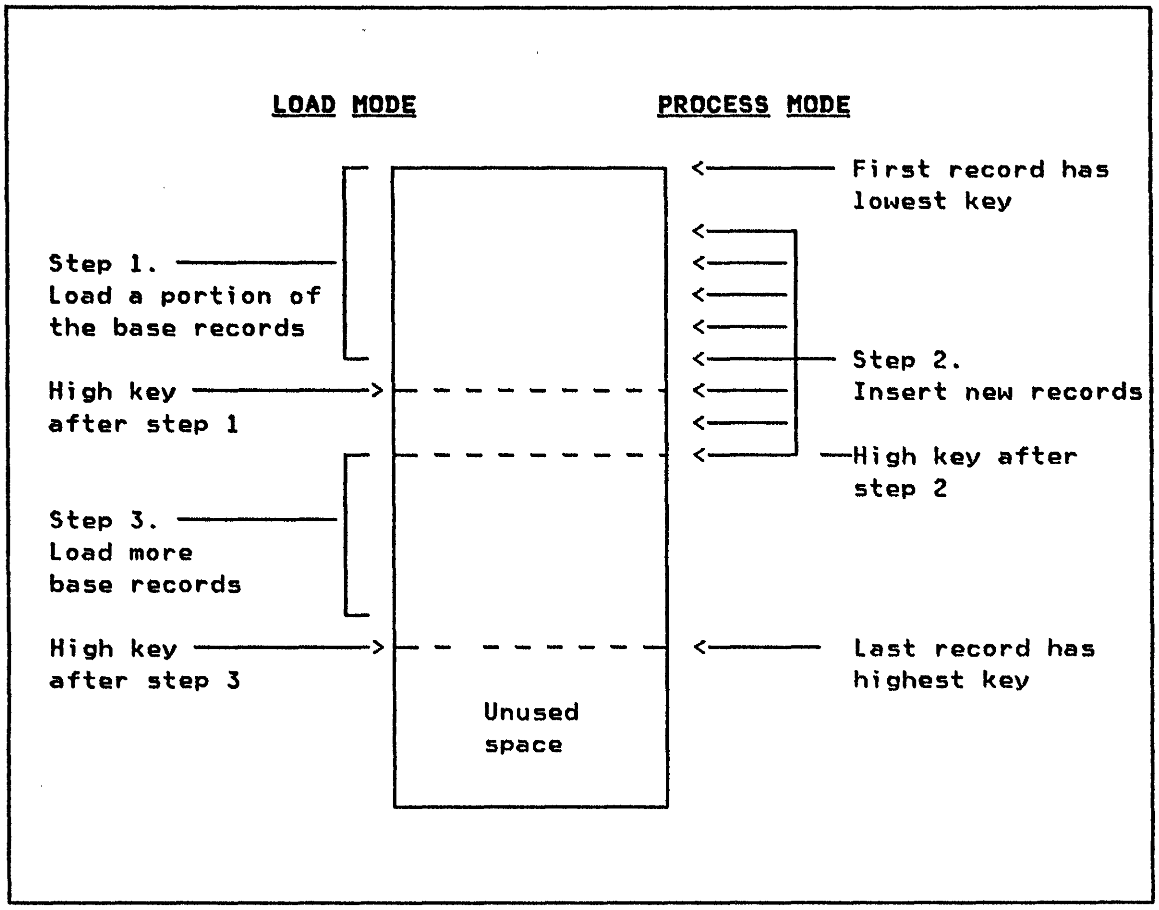

LOAD MODE
PROCESS MODE
First record has lowest key
Step 1. Load a portion of the base records
Step 2. Insert new records
High key after step 1
High key after step 2
Step 3. Load more base records
High key after step 3
Last record has highest key
Unused space

Figure 10. Loading and Inserting Records

LOADING BASE RECORDS USING $IAMUT1

After the indexed file has been defined by the $IAMUT1 DF command, you can load base records from a sequential file into the indexed file. Loading the file can be done directly by responding Y to the prompt "INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION?", when defining the file, or by using the **LO** command after the file has been defined. The data in the sequential file must be in ascending order by key.

To load base records using $IAMUT1, do the following:

1. Prepare a sequential file for input to the indexed file

2. Load the sequential file into the indexed file

Preparing Input for the Indexed File

Select one of the following methods to prepare the input in a sequential file to be loaded into an indexed file:

- If your data records are 72 bytes or less, use one of the text editors to enter your data or one of the communications utilities to get the data into an Event Driven Executive sequential file. In either case, you must know the record format used by the utility. The utilities put two 80-byte records in each 256-byte Event Driven Executive record. The first record begins at location 1, and the second record begins at location 129. This results in a blocked sequential file which can be used to load the indexed file when using the LO command of $IAMUT1. (A detailed description of the LO command is under "LO - Load Indexed File" on page IG-162.) Specify 128 for the input record length and 256 for the input block size.

- If your records have more than 72 bytes of data, you must create a program that accepts the data records and writes them to a disk, diskette, or magnetic tape file.

The data must be in ascending order, based upon the field you use as the key.

Loading an Indexed File from a Sequential File

The procedure for loading an indexed file from a sequential file is:

1. Invoke $IAMUT1 using the system command $L.

2. If you want a hard copy of the terminal prompts and responses, enter an EC command. Respond to the prompt with a Y. This will print all further prompts and responses of $IAMUT1 on the $SYSPRTR device and your terminal. If a hard copy is not required, omit this step.

3. Enter the LO command.

   Respond to the following prompts with your data set information.

```
ENTER COMMAND (?): LO
LOAD ACTIVE
ENTER OUTPUT DATASET (NAME,VOLUME):
$FSEDIT FILE RECSIZE = 128
INPUT RECORD ASSUMED TO BE     80 BYTES. OK?:
ENTER INPUT BLOCKSIZE (NULL = UNBLOCKED):
ENTER INPUT DATASET (NAME,VOLUME):
LOAD IN PROCESS

END OF INPUT DATASET
ANY MORE DATA TO BE LOADED?: N
     6 RECORDS LOADED
LOAD SUCCESSFUL
```

4. Enter the EN command to end $IAMUT1. Your program is now loaded and you can process the data with your application program.

LOADING BASE RECORDS FROM AN APPLICATION PROGRAM

Base records are records placed into an indexed file in ascending new high key sequence. That is, if a record added to the file has a key higher than any other record in the file, it is placed in a base record slot. Base records are placed in the base record slots reserved for them by use of the BASEREC parameter. You can use either the $IAMUT1 LO command or an application program to load the base records.

Base records must be loaded in ascending order by key. If you are writing your own program to load the file, use a LOAD request to connect the file to load base records. Then issue a PUT for each record. When the desired records have been loaded, issue a DISCONN request to terminate the load procedure. The only requests that can follow a LOAD request are: PUT, EXTRACT, and DISCONN.

You can also insert base records in process mode by using a PROCESS request to connect the file, followed by a PUT request for each record to be loaded. Loading records in load mode with an application program is discouraged because of slower performance.

Unless the base record loading program is written in COBOL, it need not load all base records at one time. A file that already contains records can be reconnected to load more records, but the key of each new record must be higher than any key already in the file.

COBOL programs must either load all the base records in load mode at once (because only one use of load mode is allowed on a given file) or insert the records in process mode as needed.

The limit on base records as specified on the SE command of the Indexed Access Method utility program ($IAMUT1) cannot be exceeded. If you attempt to load a record after the last allocated record area has been filled, an end-of-file condition occurs.

LOADING BASE RECORDS FROM A SEQUENTIAL FILE IN RANDOM ORDER

In order to load base records from a sequential file where keys are in random order, code an EDL program to open the indexed file in load mode. Load the SORT/MERGE program with an output exit routine specified. Write (PUT) each record to the indexed file as it is received in the output exit routine from SORT/MERGE. The output exit routine can also screen out duplicates or other unwanted records. For information on using the SORT/MERGE Program Product, refer to Sort/Merge Programmer's Guide.

CHAPTER 5. BUILDING A SECONDARY INDEX

Indexed files, like most data record files, can be a common base for many applications. You can assign **secondary keys** in your indexed files for greater flexibility in accessing records in indexed files.

Secondary keys are accessed through a **secondary index** (a separate file). Your application program requests records by their secondary key and secondary index file name. The secondary index is used to retrieve the record by its secondary key from the primary index file.

You can have more than one secondary index for a given primary index file. In order for the Indexed Access Method to know the relationships between secondary indexes and primary index files, you must create and maintain a **directory** with that information.

SECONDARY KEYS

Secondary keys are not required to be unique; different records in an indexed file can have the same key values in their secondary key field.

The secondary key can be any field within your data record that you select, however, it must meet the following requirements:

- the selected field must start at the same location in each record
- all portions of the key field must be contiguous
- the secondary key length cannot exceed 250 bytes

In a secondary index, the Indexed Access Method assigns a **sequence number** to each secondary key. The sequence number shows the sequence of loading or inserting secondary index entries.

A sample layout of a secondary index record follows:

Secondary Key	Sequence Number	Primary Key	Relative Block Number (primary)
SMITH	0001	12345AB	RBN

THE DIRECTORY

In order for the Indexed Access Method to know the relationships between secondary indexes and primary index files, you must create and maintain a

<u>directory</u> with that information. The directory describes all indexed files in the system which are either secondary indexes, or primaries which have secondary indexes associated with them. Primary index files which do not have secondary indexes associated with them are not in the directory. Use the $IAMUT1 utility to create and maintain the directory.

The directory name is $IAMDIR and it resides on the IPL volume.

The directory contains one or more groups of entries. Each group begins with an entry for the primary file and is followed by an entry for each secondary index which references that primary file.

You have the responsibility of maintaining the directory using the $IAMUT1 utility.

Each entry in the directory contains the following information:

- file name
- volume name
- primary file or secondary index indicator
- independent indicator
- invalid index indicator (secondary entry only)
- automatic update indicator (secondary entry only)

FILE NAME: The file name is the data set name supplied when the primary index file or secondary index entry is inserted in the directory.

VOLUME NAME: The file location is the volume label name where the index resides that this entry is for.

INDEPENDENT PROCESSING INDICATOR: Each entry in the directory contains an independent indicator. Independent means that the file is to be treated as an independent file without regard to associated primary or secondary files. If the independent indicator is set on for a file that is explicitly opened, the automatic update indicator is ignored.

In the case of a secondary index, this means that records retrieved are internal secondary index records, not data records from the primary file. In addition, independent means that any modification to the file (either primary or secondary) will not be reflected in its associated files. Also any changes made in a secondary index will not be reflected in the associated primary or other secondary index files.

In the case of a primary entry, any modification to the primary file will not be reflected in the associated secondary index files.

INVALID INDICATOR: The invalid indicator is initially turned on in the directory, by the directory function of $IAMUT1, when the secondary entry is inserted in the directory.

A secondary index entry is marked invalid until the secondary index has been loaded.

The load function of the utility turns off the invalid indicator.

If you build the secondary index with an application program, you must also turn off the indicator. The UE subcommand of the DR function in $IAMUT1 is used to turn off the invalid indicator, after you have successfully loaded your secondary index.

AUTOMATIC UPDATE INDICATOR: Each secondary index entry in the directory contains an automatic update indicator. Any modification to the primary file (either directly or through any secondary index activity) results in an automatic update to all secondary indexes whose automatic update indicator in the directory was specified with Y. Thus, a secondary index flagged as auto-update can be thought of as "dynamic." Each secondary index remains open until all users of it have closed. However, if the independent indicator is set on for a file that is explicitly opened, the automatic update indicator is ignored.

If the automatic update indicator was specified as N, changes are not reflected in that secondary index. This would be a "static" index. The assumption is that a static index would be rebuilt periodically to bring it up to date.

ALLOCATING AND INSERTING ENTRIES IN A DIRECTORY

Although the Indexed Access Method references the directory, it never modifies the directory. The one function that is performed on the directory automatically is that the secondary load option sets the invalid indicator off following successful completion.

To define the existence of a secondary index, use $IAMUT1 to perform the following two steps:

1. allocate a directory using the DR (directory function) of $IAMUT1

2. establish the fact that a secondary index will exist by making an entry in the directory using the IE (insert entry) command of $IAMUT1

Remember that primary index file entries precede their associated secondary index entries. The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1]  ENTER COMMAND (?): DR
```

[1] Entering DR causes the following prompt sequence.

```
[2]  ENTER DIRECTORY COMMAND (?): AL
[3]  MAX # OF DIRECTORY ENTRIES:  10
[4]  THE DIRECTORY DS REQUIRES    1 EDX RECORDS, CONTINUE (Y|N|EN)? Y

[5]  DIRECTORY DATA SET ALLOCATED: $IAMDIR,EDX002
```

[2] Responding to this prompt with **AL** (allocate) causes a directory allocation sequence to begin.

> **Note:** The allocation sequence is only required the first time you set up secondary indexes. Future entries can be added using (IE) insert directory function.

[3] Reply with maximum number of directory entries you want allocated for the directory. You will need one entry for each secondary index and one entry for each primary that has a secondary index associated with it. A null response will allocate the maximum (default) of 47 entries.

[4] Based on your previous response, the size of the required directory is calculated and you are informed of the number of EDX records required to allocate your requested directory. You are also given three options as follows:

1. Y - the opportunity to continue the directory allocation

2. N - do not allocate the directory; allow me to change the size of the directory

3. EN - end the allocate function; return to [1] of the DR function of $IAMUT1 to enter another command.

[5] Because [Y] was replied, the directory is allocated. If the directory is allocated successfully, you are informed that is has been allocated, the name of the directory of course is $IAMDIR, and the IPL volume where it is always allocated is displayed.

> **Note:** The allocation sequence is only required the first time you set up secondary indexes.

The prompt sequence continues.

```
[6]  ENTER DIRECTORY COMMAND (?):  IE

[7]  (DSNAME,VOLUME): EMP#,EDX002
[8]  IS THIS A SECONDARY ENTRY (Y/N) N

[9]  DIRECTORY INSERT SUCCESSFUL

[10] ENTER DIRECTORY COMMAND (?): IE
```

[6] Replying **IE** (insert entry) allows you to insert entries into the directory. A primary entry must be inserted before its associated secondary index entries.

> **Note:** Primary files may exist at this time, however, secondary indexes cannot be created until an entry for it has been inserted in the directory.

[7] Your data set name and volume name where your primary index file or secondary index resides. The volume name is not required if the data set is on system volume such as EDX002.

[8] This prompt lets $IAMUT1 know whether to set the primary or secondary entry indicator. Reply Y for a secondary index entry, or N for a primary index entry.

[9] This message informs you that the entry has been successfully inserted into the directory.

[10] At this point you can end the directory function by responding to the prompt with EN, or reply any other directory function.

Because this was a primary entry we can now respond with IE and insert secondary directory entries. In this case, secondary entries are being made and that is why we responded with **IE** and caused the prompts to continue as follows:

```
[11]  (DSNAME,VOLUME): NAME,EDX002
[12]  IS THIS A SECONDARY ENTRY (Y/N)? Y

[13]  ASSOCIATED PRIMARY ENTRY (DSNAME,VOLUME): EMP#,EDX002
[14]  AUTO-UPDATE (Y/N)? Y

[15]  DIRECTORY INSERT SUCCESSFUL

[16]  ENTER DIRECTORY COMMAND (?): IE
```

[11] The secondary index data set name is NAME on volume EDX002 and therefore, the volume name could have been omitted.

[12] Because this is an entry for a secondary index, the correct reply is Y. At this point in the prompt sequence the prompts change from the previous sequence because of the positive reply, Y.

[13] The associated primary entry data set name, which the previous entry sequence (5 - 9) was for, is EMP#,EDX002. This is the point where the secondary indexes establish their association to the primary index files for which the secondary index is built.

[14] The response to this prompt establishes whether automatic update option is to be effective for this secondary index. For a description of automatic update, see "Automatic Update Indicator" on page IG-69. The recommended response is Y, also if a null entry is supplied, the default is Y (yes).

[15] You are informed when the insert is successfully completed.

[16] As seen previously, you again have the option of selecting another directory function. In this description, IE was again selected to insert the following two secondary index entries.

A second secondary index entry named CITY,EDX002, is inserted for the associated primary index file named EMP#.

```
(DSNAME,VOLUME): CITY,EDX002
IS THIS A SECONDARY ENTRY (Y/N) Y

ASSOCIATED PRIMARY ENTRY (DSNAME,VOLUME): EMP#,EDX002
AUTO-UPDATE (Y/N)? Y

DIRECTORY INSERT SUCCESSFUL

ENTER DIRECTORY COMMAND (?):  IE
```

A third secondary index entry named LEVEL,EDX002, is inserted for the associated primary index file named EMP#.

```
(DSNAME,VOLUME): LEVEL,EDX002
IS THIS A SECONDARY ENTRY (Y/N) Y

ASSOCIATED PRIMARY ENTRY (DSNAME,VOLUME): EMP#,EDX002
AUTO-UPDATE (Y/N)? N

DIRECTORY INSERT SUCCESSFUL
```

The following example uses a different directory function: LE (list directory entries). This example shows the directory which was just allocated and four entries inserted; one primary and three secondaries.

```
[1]  ENTER DIRECTORY COMMAND (?): LE

[2]  ENTRY (DSNAME,VOLUME) BLANK=ALL:

[3]                  PRIMARY  INDE-              AUTO
     DSNAME  VOLUME DATA SET PENDENT  INVALID  UPDATE

[4]  EMP#    EDX002 YES      NO       ****     ****
[5]  NAME    EDX002 NO       NO       YES      YES
[6]  CITY    EDX002 NO       NO       YES      YES
[7]  LEVEL   EDX002 NO       NO       YES      NO

[8]  NUMBER OF DIRECTORY ENTRIES USED =     4
[9]  NUMBER OF AVAILABLE ENTRY SLOTS =     6
     DIRECTORY LIST COMPLETED
```

[1] The DR (directory) subcommand LE prints specified directory statistics.

[2] Respond to this prompt with the specific data set name and volume you wish the statistics listed for, or press the Enter key with no DSNAME or VOLUME name specified to list the entire directory. This request is for all entries in the directory which was just allocated and inserts made in the previous examples.

[3] Column headings for the listed information from the directory showing the following information:

- data set name that the statistics are for
- volume name where the data set resides
- whether this is a primary or secondary index
- is the independent indicator on for the named data set (yes or no)
- is the invalid index indicator on for the named data set (yes or no)
- is the auto-update indicator on for the named data set (yes or no)

[4] For the primary index file (data set) named EMP#, on volume EDX002, the independent indicator is off, there is no invalid indicator for a primary file, there is no auto-update indicator for a primary file. Modifications are always made to the primary index file if the independent indicator is not on.

[5] For the secondary index named NAME, on volume EDX002, the independent indicator is off, the invalid index indicator is on because the index has not been loaded, the auto-update indicator is on as requested when the entry for this secondary index was inserted.

[6] Same statistics as previous data set.

[7] For the secondary index named LEVEL, on volume EDX002, the independent indicator is off, the invalid indicator is on (index is not loaded), and the auto-update indicator is off.

[8] There were 10 entries allocated and 4 inserts (one primary and three secondaries).

[9] The resulting empty directory slots for additional inserts is six.

SECONDARY INDEX

Depending upon your need, you may have one or several secondary indexes for a given primary index file. A secondary index is built for a specific primary index file and cannot be used with any other file. Each secondary index is a separate Indexed Access Method file.

Application programs accessing indexed records by their secondary key are required to open the secondary index and access the records using the secondary key. When primary index records are updated, inserted or deleted, some or all secondary indexes associated with that primary index file can be updated automatically by the Indexed Access Method, according to the options selected when the secondary index directory is set up.

Setting up a Secondary Index

To provide access by a secondary key, you must build a secondary index. The secondary index must have a unique file name.

To set up a secondary index, you must do the following using $IAMUT1:

1. create the secondary index

2. load the secondary index

DEFINING AND LOADING A SECONDARY INDEX

Your secondary index should be structured so that the base records parameter is equal to or greater than the number of records in the primary index file. This will assure that when you build your secondary index, it will be large enough to hold at least as many records as there are in the primary index file.

> **Note:** If the associated primary index file, for which the secondary is being defined, is an existing Version 1 created file, you must use $VERIFY to update the record counts before defining the secondary file.

The key size and key position specified for the secondary index must be the key size and starting position of the secondary key within the primary index record.

You can create a secondary index the same way you create a primary index file, using the $IAMUT1 utility SE (and DF) commands. The utility prompts you requesting whether the secondary index being defined is also to be loaded. If YES is specified, the utility does the following:

1. creates the secondary index but does not format it

2. opens the primary file, reads the records sequentially, and extracts the primary and secondary keys from each record, retaining the relative data block address (RBN) of each record

3. invokes the Sort/Merge Program Product to sort by secondary key (and by primary key within secondary)

4. opens the secondary index, formats the sorted keys, their sequence numbers which are now assigned, and the relative data block addresses of the primary file data records into blocks

5. writes the blocks into the secondary index

Before a secondary index can be loaded, it must have been defined using $IAMUT1. A secondary index can be deleted, then created and loaded again at any time. If a primary file has more than one secondary index, each must be created and loaded separately.

$IAMUT1 Option Selection Guide

Having read the preceding material, you are probably ready to make a choice as to which option you want to use in defining your secondary index. The following table will help you to find the appropriate information, based on your secondary index defining objectives.

Your Objective	Option	Information location
You want the Indexed Access Method to calculate and structure your index	Option 1	See "Option 1" on page IG-77
You want to structure your secondary index using specific parameters	Option 2	See "Option 2" on page IG-79
You want to structure your secondary index using the parameters of an existing secondary index	Option 3	See "Option 3" on page IG-82

EXAMPLE 1: DEFINING A SECONDARY INDEX USING $IAMUT1

The Indexed Access Method utility, $IAMUT1, option 1, provides you with the opportunity to select only those parameters necessary to set up a secondary index.

OPTION 1

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1] ENTER COMMAND (?): SE
```

[1] Entering SE causes the following option list prompt to be displayed.

```
    SET FILE DEFINITION PARAMETERS
    0 = EXIT
[2] 1 = SIGNIFICANT PARAMETERS
    2 = ALL PARAMETERS
    3 = PARAMETERS FROM EXISTING INDEXED DATA SET
    ENTER OPTION: 1
```

[2] Respond to this prompt by entering the digit '1'. This response causes a one line prompt from the next prompt sequence to be displayed.

Note: Although the following prompts are displayed one line at a time when using the utility, all the prompts are listed here in logical groups for simplicity in describing the parameters.

```
[3]  SECONDARY INDEX  (Y/N)?:  Y
[4]  ENTER SECONDARY DATASET  (NAME,VOLUME):  CITY,EDX002
[5]  SECONDARY KEY SIZE     :4
[6]  SECONDARY KEY POSITION :5

     DATA SET SIZE IN EDX RECORDS:              10
     INDEXED ACCESS METHOD RETURN CODE:         -1
     SYSTEM RETURN CODE:                        -1
     CREATE/DEFINE FILE (Y/N)?: Y
     NEW DATA SET IS ALLOCATED
     DO YOU WANT IMMEDIATE WRITE-BACK? N
     INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION? L
     DEFINE IN PROGRESS
     DATA SET SIZE IN EDX RECORDS:              10
     INDEXED ACCESS METHOD RETURN CODE:         -1
     SYSTEM RETURN CODE:                        -1
     PROCEED WITH LOAD/REORGANIZE (Y/N)? Y
[7]  SECONDARY INDEX LOAD ACTIVE
     ANSWER NULL FOR ALLOCATING DEFAULT WORK DATASET $SORTWRK
[8]  SORT WORK DATASET (DSNAME,VOLUME):
     SORT WORK DATASET REQUIRES      20 EDX RECORDS
             5 RECORDS LOADED
     SECONDARY LOAD SUCCESSFUL
```

[3] Reply Y to this prompt because you are defining a secondary index.

[4] Enter the data set name and volume where this index is being defined.

[5] Specify the length of the secondary key within the primary index record for which this index is being defined.

[6] Specify the starting position of the secondary within the primary index file record. The first byte of the record is number 1.

[7] The secondary index load function is active.

[8] At this point there are three possible responses:

1. A null response, just pressing the enter key, will cause $SORTWRK to be allocated on the IPL volume, if space is available. The size of the data set is calculated by the utility. If $SORTWRK already exists, this indicates that another user is using the default work data set and you will be prompted again for a work data set name.

2. Entering a comma (,) followed immediately with a volume name, then pressing the enter key, causes the utility to try to allocate $SORTWRK on the specified volume.

3. Entering a data set name and optionally a volume name (no volume name entered causes the IPL volume to be used) causes the utility to calculate the size of data set required and allocate it according to your response.

Notes:

1. If $IAMUT1 allocates the data set for you, the data set will be automatically deleted at the end of the sort operation. However, if you provide either the name of an already existing data set (other than $SORTWRK) or a data set name you want $IAMUT1 to allocate, the data set will not be deleted at the end of the sort.

2. The sort work data set cannot always be calculated precisely because the size is dependent on several variables related to the input file. In most cases the calculated size will be adequate. However, if the size calculated is too small, the sort will end prematurely. If this happens you can preallocate a data set with a larger size than that calculated by $IAMUT1 and execute the sort again.

OPTION 2

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command **$L $IAMUT1**.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1] ENTER COMMAND (?): SE
```

[1] Entering **SE** causes the following option list prompt to be displayed.

```
    SET FILE DEFINITION PARAMETERS
    0 = EXIT
    1 = SIGNIFICANT PARAMETERS
[2] 2 = ALL PARAMETERS
    3 = PARAMETERS FROM EXISTING INDEXED DATA SET
    ENTER OPTION: 2
```

[2] Respond to this prompt by entering the digit '2'. This response causes a one line prompt from the next prompt sequence to be displayed.

Note: Although the following prompts are displayed one line at a time when using the utility, the entire prompt list is shown for simplicity in presentation.

```
[3]  SECONDARY INDEX  (Y/N)?:  Y
[4]  ENTER SECONDARY DATASET (NAME,VOLUME):  NAME,EDX002
[5]  PARAMETER  DEFAULT NEW VALUE
     BASEREC         20:20
     BLKSIZE        256:
     KEYSIZE          4:
     KEYPOS           5:
     FREEREC          0:
     FREEBLK          0:
     RSVBLK        NULL:
     RSVIX            0:
     FPOOL         NULL:
     DELTHR        NULL:
     DYN           NULL:05
     TOTAL LOGICAL RECORDS/DATA BLOCK:         15
     FULL RECORDS/DATA BLOCK:                  15
     INITIAL ALLOCATED DATA BLOCKS:             2
     INDEX ENTRY SIZE:                         12
     TOTAL ENTRIES/INDEX BLOCK:                20
     FREE ENTRIES/PIXB:                         0
     RESERVE ENTRIES/PIXB(BLOCKS):              0
     FULL ENTRIES/PIXB:                        20
     RESERVE ENTRIES/SIXB:                      0
     FULL ENTRIES/SIXB:                        20
     DELETE THRESHOLD ENTRIES                   0
     FREE POOL SIZE IN BLOCKS                   5
     # OF INDEX BLOCKS AT LEVEL   1:            1
     DATA SET SIZE IN EDX RECORDS:             10
     INDEXED ACCESS METHOD RETURN CODE:        -1
     SYSTEM RETURN CODE:                       -1
     CREATE/DEFINE FILE (Y/N)?:  Y
     DATA SET EXISTS ALREADY AND IS LARGE ENOUGH
     DO YOU WISH TO USE IT (Y/N): Y
     DO YOU WANT IMMEDIATE WRITE-BACK?  N
     INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION? L
     DEFINE IN PROGRESS
     DATA SET SIZE IN EDX RECORDS:             10
     INDEXED ACCESS METHOD RETURN CODE:        -1
     SYSTEM RETURN CODE:                       -1
     PROCEED WITH LOAD/REORGANIZE (Y/N)? Y
[6]  SECONDARY INDEX LOAD ACTIVE
     ANSWER NULL FOR ALLOCATING DEFAULT WORK DATASET $SORTWRK
[7]  SORT WORK DATASET (DSNAME,VOLUME):
     SORT WORK DATASET REQUIRES      20 EDX RECORDS
             5 RECORDS LOADED
     SECONDARY LOAD SUCCESSFUL
```

[3] Reply Y to this prompt because you are defining a secondary index.

[4] Enter the data set name and volume where this index is being defined.

[5] The following parameter list allows you to precisely define the secondary index structure.

[6] The secondary index load function is active.

[7] If the name of a data set and volume are entered, Sort/Merge will use it for the work data set. If a null response is made, the utility will calculate the size data set required and allocate it with the name $SORTWRK on the IPL volume.

Notes:

1. For a more complete description of the responses available and the possible conditions that could exist, see step 8 description under "Option 1."

2. The following messages are from the IBM Sort/Merge Program Product, program number 5719-SM2. The following message list is the result of the secondary load function calling and executing Sort/Merge. For a description of the Sort/Merge program and its messages refer to <u>IBM Series/1 Event Driven Executive Sort/Merge Programmer's Guide</u>, SL23-0016..

```
SORT099N ----+----1----+----2----+----3----+----4----+----5----+-
SORT000* LO$SYSPRTR
SORT001P SORT/MERGE SPECIFICATION PHASE STARTED
SORT000* HSORTR  12A0DP
SORT000* DW $SORTWRKTVOL
SORT000* FNC0001  12
SORT000* FR
SORT075P SPECIFICATION PHASE ENDED
SORT076P INPUT PHASE STARTED
SORT082P INPUT PHASE ENDED            5           1           1
SORT085P FINAL MERGE PHASE STARTED           1      4     5
SORT086P FINAL MERGE PHASE ENDED
SORT088P RECORDS READ FROM INPUT DATA SET(S):                0
SORT089N RECORDS INSERTED BY INPUT EXIT ROUTINE:             5
SORT090N RECORDS DELETED BY INPUT EXIT ROUTINE:              0
SORT091N RECORDS INSERTED BY OUTPUT EXIT ROUTINE:            0
SORT092N RECORDS DELETED BY OUTPUT EXIT ROUTINE:             0
SORT093N RECORDS WRITTEN TO OUTPUT DATA SET:                 0
SORT094N I/O ERRORS ACCEPTED:                                0
SORT095N I/O ERRORS SKIPPED:                                 0
SORT149N RECORDS SORTED OR MERGED:                           5
SORT097P NORMAL ENDING FOR SORT/MERGE PROCESSING
```

OPTION 3

The $IAMUT1 Indexed Access Method utility can be loaded with the Event Driven Executive operator command $L $IAMUT1.

When $IAMUT1 is loaded the first prompt displayed is as follows:

```
[1] ENTER COMMAND (?): SE
```

[1] Entering SE causes the following option list prompt to be displayed.

```
    SET FILE DEFINITION PARAMETERS
    0 = EXIT
    1 = SIGNIFICANT PARAMETERS
    2 = ALL PARAMETERS
[2] 3 = PARAMETERS FROM EXISTING INDEXED DATA SET
    ENTER OPTION: 3
```

[2] Respond to this prompt by entering the digit '3'. This response causes a one line prompt from the next prompt sequence to be displayed.

Note: Although the following prompts are displayed one line at a time when using the utility, the entire prompt list is shown for simplicity in describing the parameters.

```
[3]  SECONDARY INDEX  (Y/N)?:  Y
[4]  ENTER SECONDARY DATASET  (NAME,VOLUME):  LEVEL,EDX002
[5]  NAME OF EXISTING INDEXED DATA SET (NAME,VOLUME): CITY,EDX002
[6]  NEW PARAMETERS EXACTLY SAME AS ORIGINAL PARAMETERS (Y/N)? Y
     DATA SET SIZE IN EDX RECORDS:               10
     INDEXED ACCESS METHOD RETURN CODE:          -1
     SYSTEM RETURN CODE:                         -1
     CREATE/DEFINE FILE (Y/N)?: Y
     NEW DATASET IS ALLOCATED
     DO YOU WANT IMMEDIATE WRITE-BACK? Y
     INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION? L
     DEFINE IN PROGRESS
     DATA SET SIZE IN EDX RECORDS:               10
     INDEXED ACCESS METHOD RETURN CODE:          -1
     SYSTEM RETURN CODE:                         -1
     PROCEED WITH LOAD/REORGANIZE (Y/N)? Y
     SECONDARY INDEX LOAD ACTIVE
     ANSWER NULL FOR ALLOCATING DEFAULT WORK DATASET $SORTWRK
[7]  SORT WORK DATASET (DSNAME,VOLUME):
     SORT WORK DATASET REQUIRES        20 RECORDS
               5 RECORDS LOADED
     SECONDARY LOAD SUCCESSFUL
```

[3] Reply Y to this prompt because you are defining a secondary index.

[4] Enter the data set name and volume where this index is being defined.

[5] Enter the data set name and volume of the secondary index whose parameters are to be used for this index.

[6] If all parameters are to be the same as those initially set for the data set name entered in prompt [3], reply Y. However, if you want the parameters adjusted, based on current file status, reply N.

[7] If the name of a data set and volume are entered, Sort/Merge will use it for the work data set. If a null response is made, the utility will calculate the size data set required and allocate it with the name $SORTWRK on the IPL volume.

Note: For a more complete description of the responses available and the possible conditions that could exist, see step 8 description under "Option 1."

LOADING A SECONDARY FILE WITH AN APPLICATION PROGRAM

You have the option of allowing $IAMUT1 to load your secondary file at the time it is created, as was demonstrated in "Option 2" on page IG-79, or you can load it with an application program. The sequence of operation for loading your secondary index with an application program is described here.

A secondary file has the following format:

Secondary Key	Sequence Number	Primary Key	Relative Block Number
SMITH	0001	12345AB	RBN
	4 bytes		4 bytes

In preparation for loading your secondary index, allocate the following sort data sets. The size of the input and output data sets are calculated using the lengths of keys from the primary index file record plus four bytes for the sequence number and four bytes for the RBN.

Secondary key length + primary key length + 8 bytes

- Sort input data set
- Sort output data set
- Sort work data set

If you have the Sort/Merge licensed program product, program number 5719-SM2, refer to the IBM Series/1 Event Driven Executive Sort/Merge Programmer's Guide, SL23-0016, for details of the sort work data set and sort specification data sets. Otherwise, use your own sort program.

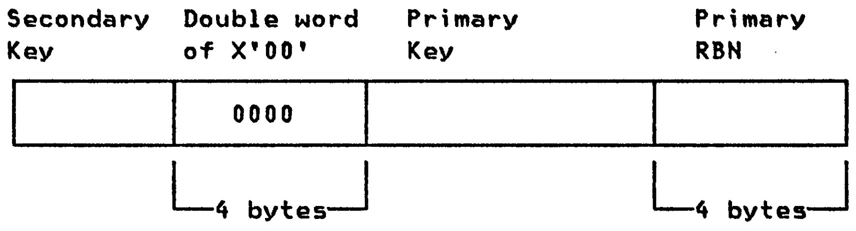
Loading your secondary index requires the following sequence:

1. Open the primary index file in process mode

2. Retrieve a primary index file record with a GETSEQR request (retrieves the primary record plus the RBN)

   a. When end-of-data condition is reached, go to step 6

3. Using values from the retrieved record, build a secondary record with the following format:



4. Move the newly built secondary record into the sort input data set.

5. Return to step 2 and repeat the sequence.

6. Sort the records in the sort input data set using the following sort specifications.

   a. Sort the input records on position 1 through n-4 into ascending sequence (n= the length of the records as calculated previously for the sort data sets).

7. Open the secondary index, which is to be loaded, in load mode.

   If your program is written in Event Driven Language (EDL), specify the independent option when you open the secondary index. If you are using a high level language, use $IAMUT1 to turn on the independent indicator.

8. Read a record from the sorted output.

   a. When end-of-file is reached, go to step 14.

9. Move a sequence number into the retrieved record's sequence number field (use X'0000' for the first record).

10. Increment the sequence number by a value of 1.

11. Use a PUT request to load the record into the secondary file.

12. Return to step 8 and repeat the sequence.

13. Issue a DISCONNECT to the primary index file and secondary index.

14. Using the $IAMUT1 utility, turn off the invalid indicator for this secondary index entry in your directory. Also, turn off the independent indicator if you truned it on in step 7.

CHAPTER 6. PROCESSING THE INDEXED FILE

This chapter provides information for designing applications that use the Indexed Access Method. It contains information about:

- Connecting and disconnecting the indexed file
- Accessing the indexed file
- Maintaining the indexed file

"Chapter 7. Coding the Indexed Access Method Requests" contains a detailed description of the EDL coding syntax of each Indexed Access Method request. You may wish to refer to it while reading this chapter.

CONNECTING AND DISCONNECTING THE INDEXED FILE

An indexed file must be defined and formatted by using the $IAMUT1 utility set parms (SE) and define (DF) commands before issuing a LOAD or PROCESS request to the file.

Prior to using an indexed file, you must issue either a LOAD or PROCESS request to connect it to your program. The file must be defined in your PROGRAM statement or by a DSCB statement. A CALL statement specifying either LOAD or PROCESS automatically opens the Indexed Access Method file. If you have an already open DSCB for the Indexed Access Method file you can pass it as a parameter, but that is not required.

CONNECTING

A LOAD or PROCESS request builds an indexed access control block (IACB) that is associated with an indexed file. The IACB connects a request to the file.

In load mode, data records are placed in the file sequentially (free records and blocks are skipped). When in process mode, data records are placed in the first appropriate slot in the file (free space is used) unless the record has a new high key. In the case of a new high key, the record is placed in the next available base record slot.

Only one LOAD request can be active for a given file. However, processing can take place concurrently with loading.

Multiple IACBs can be associated with the same file. Data integrity is maintained by a locking system that assigns file locks, record locks, or block locks to the requesting IACB. This prevents concurrent modification of index or data records, thereby avoiding the possibility of a double update situation.

Some applications will not need to wait for a lock to be released on a record, block, or buffer. In these situations you might want to use the conditional requests available for some Indexed Access Method functions. The conditional function requests allow control to be returned immediately to the requesting program for other processing, then return later to attempt to retrieve the record which was locked. The conditional requests are described in "Chapter 7. Coding the Indexed Access Method Requests."

An IACB can hold only one lock at a time; if your application requires concurrent execution of functions that obtain locks (direct update or sequential update - see "Accessing the Indexed File" on page IG-89 for a description of these functions), you must issue multiple PROCESS requests to build multiple IACBs.

DISCONNECTING

A DISCONN request disconnects an IACB from the file, releases the storage for that IACB, releases locked blocks or records being held by that IACB, and writes out to disk any blocks that are being held in the buffer. The DISCONN request can be issued at any time during loading or processing.

There is no automatic DISCONN on task termination. Failure to disconnect your indexed files prior to task termination may prevent resources that were allocated to your task from being allocated to other tasks and updated records from being written to your file.

Using Secondary Keys

To access a file by a secondary key, you issue either a LOAD or PROCESS request, specifying the file name of the **secondary index** and specifying **secondary keys** when referencing data records. The Indexed Access Method determines the relationships among the files by using the directory and automatically opens the primary file. All subsequent operations done under this LOAD or PROCESS access the file using the secondary index. You must open a file by the primary name to access it by the primary keys.

Direct retrievals use the secondary index, and sequential retrievals return records in order by secondary key. Records within a group which have the same secondary key are returned in the order which the records were written into the file. Each application must determine whether the correct record has been retrieved when duplicate keys are possible; the Indexed Access Method provides no facility for that determination.

When records are updated, inserted, or deleted, in primary index files, some or all secondary indexes can be updated automatically according to the options you selected in the directory entries. These options are: auto-update and independent processing.

If the auto-update indicator is on in the directory entry for a secondary index and you open the associated primary file to insert, delete, or update records, the associated secondary index will be updated automatically. There is no consideration for whether the independent indicator is on for the secondary. However, if the invalid indicator for the secondary entry is on, the secondary index is not updated.

The independent indicator is used when a secondary index is opened in load mode to add new entries to the file.

> **Note:** When records are accessed by their secondary key, you must ascertain through your application program that you have retrieved the correct record because of the possibility of duplicate keys.

ACCESSING THE INDEXED FILE

Initiate general purpose access to an indexed file with a PROCESS request. After the PROCESS request has been issued, any of the following functions can be requested:

- Direct reading - Retrieving a single record independently of any previous request.

- Direct updating - Retrieving a single record for update; complete the update by either replacing, deleting, or releasing the record.

- Sequential reading - Retrieving the next logical record relative to the previous sequential request.

  The first sequential request can access the first record in the file or any other record in the file by key (except COBOL applications).

- Sequential updating - Retrieving the next logical record for update; complete the update by either replacing, deleting, or releasing the record.

- Inserting - Placing a single record, in its logical key sequence, into the indexed file.

- Deleting - Removing a single record from the indexed file.

- Extracting - Extracting data that describes the file.

Note that the update functions require more than one request.

When a function is complete, another function may be requested, except that a sequential processing function may be followed only by another sequential function. You can terminate sequential processing at any time by issuing a DISCONN or ENDSEQ request. An end-of-data condition also terminates sequential processing.

DIRECT READING

Use the GET request to read a record using direct access. The key parameter is required and must be the address of a field of full key length regardless of the key length specification.

The record retrieved is the first record in the file that satisfies the search argument defined by the key and key relation (krel) parameters. The key field in your program is updated to reflect the key contained in the record that satisfied the search.

If the key length is specified as less than the full key length, only part of the key field is used for comparison when searching the file. For example, the keys in a file are AAA, AAB, ABA, and ABB, the key field contains AB0, and key relation is EQ. If key length is zero, the search argument defaults to the full key AB0 and a record-not-found code is returned. If the key length specification is 2 and the search argument is AB, the third record is returned. If the key length specification is 1 and the search argument is A, the first record is returned.

DIRECT UPDATING

To update a record using direct access:

1. Retrieve the record with a GET request, specifying the key and key relation (krel) parameters.

2. Complete the update by doing one of the following:

    - If you want to change the record, modify the record in your buffer (do not change the key field of the record). Issue a PUTUP request to return the updated record to the file.

    - If you do not want to change the record, issue a RELEASE request.

    - If you want to delete the record, issue a PUTDE request.

The key parameter must be specified as the address of a field of full key length. The primary key cannot be modified during the update; a secondary key can.

The only valid requests, other than DISCONN and EXTRACT, that can follow GET for direct update are PUTUP, PUTDE, and RELEASE.

During the update, the subject record is locked (made unavailable) to any other request until the update is complete. Even if no action is taken after the GET request is issued, the RELEASE request is required to release the lock on the record. You may wish to use the conditional option on your requests to avoid unnecessary wait for locks. For conditional request coding see "Chapter 7. Coding the Indexed Access Method Requests" on page IG-97. For details on long lock time or dead lock condition, see "Deadlocks and the Long-Lock-Time Condition" on page IG-218.

SEQUENTIAL READING

Use the GETSEQ request for sequential access to records. After a sequential processing request has been initiated, only sequential functions can be requested until an end-of-data condition occurs or an ENDSEQ request is issued. Processing is terminated when a DISCONN request is issued or an error or warning is returned.

Figure 11 on page IG-92 summarizes the protocol for sequential processing.

Note: You can sequentially process a file more than once.

To begin sequential access with the first record in a file, set the key address to zero. To start with any other record, specify a search argument by specifying the key and key relation (krel) parameters.

If you specify a search argument, the key field is modified to reflect the key of the first record found.

After the first retrieval, a GETSEQ retrieves the next sequential record regardless of any key or key relation specification. Therefore, you can use the same GETSEQ statement to retrieve all records. A search argument on succeeding retrievals is ignored and the key field is not modified.

When using secondary keys, you access the duplicate keys with a sequential get request. For example:

```
     GETSEQ SMITH
```

Issuing the same request repeatedly will return all of the secondary keys whose value is SMITH. You must check to determine when the key changes, or when you have retrieved the particular record you want within that sequence of keys.

Specify ENDSEQ to stop reading before the end of data is reached. Reading ends automatically at the end of data. The end-of-data condition occurs when an attempt is made to retrieve a record after the last record in the file.

If you specify the end-of-data exit (EODEXIT) parameter on the PROCESS request, control is transferred to the address specified by the EODEXIT parameter when the end-of-data condition occurs.

During sequential reading, the block that contains the record is locked, making all records in the block unavailable to other requesters until the last record of the block is processed or sequential processing is ended. For details on long lock time or dead lock condition, see "Deadlocks and the Long-Lock-Time Condition" on page IG-218.

SEQUENTIAL UPDATING

To update a record using sequential access:

1. Retrieve the record with a GETSEQ request for update, specifying the key and one of the update key relation (krel) parameters. The key is used only on the first retrieval. Do not specify a key if processing is to begin with the first record in the file.

2. Complete the update by doing one of the following:

   - If you want to change the record, modify the record in your buffer (do not attempt to change the primary key field of the record). Issue a PUTUP request to return the updated record to the file.

   - If you do not want to change the record, issue a RELEASE request.

   - If you want to delete the record, issue a PUTDE request.

During sequential updating, the block that contains the record is locked, making all records in the block unavailable to other requesters until the last record of the block is processed or sequential processing is ended.

Terminate processing with an ENDSEQ request or a DISCONN request either before or after completing the update. Processing is also terminated on an end-of-data condition.

REQUEST/CONDITION	CAN BE FOLLOWED BY:
GETSEQ	DISCONN END-OF-DATA CONDITION ENDSEQ PUTUP PUTDE RELEASE
End-of-data condition or ENDSEQ request	DISCONN GET PUT DELETE
PUTUP	DISCONN ENDSEQ GETSEQ
PUTDE	DISCONN ENDSEQ GETSEQ
RELEASE	DISCONN ENDSEQ GETSEQ

Figure 11. Protocol for Sequential Updating

INSERTING RECORDS

To insert a new record in a file, issue a PUT request after the file has been connected with a PROCESS. The Indexed Access Method uses the primary key of the record to insert the record into the file.

The primary key of the inserted record must be different from any key in the file; otherwise, a duplicate key error occurs. The key can be higher than any key in the file.

If you are not loading base records, and want to insert records into the file in random order, the following should be satisfied:

- for files defined by option 1, ensure that random (R) was specified
- for files defined by option 2, ensure that sufficient free pool space was specified

DELETING RECORDS

Use DELETE to delete a record from the file. Specify the full key of the record. If no record exists with the specified key, a warning return code is returned.

EXTRACTING INDEXED FILE INFORMATION

The EXTRACT request provides information about a file from the file control block (FCB) or FCB Extension. It can also return data paging statistics to the calling program with an option to reset the counters. Data paging is described under "Data Paging" on page IG-206.

The FCB includes information such as key length, key displacement, block size, record size, and other data regarding the file structure. The FCB Extension contains the $IAMUT1 utility SE command parameters that were used to define the file.

The EXTRACT request copies the file control block or the FCB Extension to an area that you provide. The file must have been connected by a LOAD or PROCESS request.

The contents of the FCB block and the FCB Extension are described by FCBEQU, a unit of copy code that is supplied with the Indexed Access Method. Use COPY FCBEQU to include these equates in an EDL program.

An EXTRACT issued for a secondary file returns the primary FCB with the secondary key size and position of the secondary key. If you want the actual FCB of the secondary file, you must open the secondary file independent and then the secondary index FCB will be returned for the EXTRACT request. The FCB extension returned is always the secondary FCB extension.

MAINTAINING THE INDEXED FILE

This section describes how to maintain Indexed Access Method files. The following topics are discussed:

- File backup and recovery
- File recovery without backup
- Reorganizing the file
- Dumping the file
- Deleting the file
- Verifying an indexed file

FILE BACKUP AND RECOVERY

To protect against the destruction of data, copy the indexed file (or the volume in which the file exists) at regular intervals using the $COPY utility. (See the Operator's Reference for instructions on using the Event Driven Executive utilities.) To obtain a sequential dump of an indexed file, use the $IAMUT1 utility UN command. During the interval between making copies, you should keep a journal file of all transactions made against the indexed file.

The journal file can be a consecutive file containing records that describe the type of transaction and the pertinent data. A damaged indexed file can be recovered by updating the backup copy from the journal file.

For example, suppose an indexed file named REPORT is lost because of physical damage to the disk. The condition that caused the error has been repaired and the file must be recovered. Delete REPORT, copy the backup version of REPORT to the desired volume, and process the journal file to recreate the file.

If a data-set-shut-down condition exists, cancel $IAM and reload it. Then issue a PROCESS to the REPORT file and, using the journal file, reprocess the transactions that occurred after the backup copy was made. Refer to "The Data-Set-Shut-Down Condition" on page IG-215 for more information on the data-set-shut-down condition.

Backing Up A Secondary Index

A Secondary index can be backed up the same as primary index files. However, if your primary file is backed up you can rebuild your secondary from the backup copy of the primary indexed file.

Duplicate secondary keys are maintained in the order they are inserted by a secondary key sequence number. This sequence number is incremented with each new insert. When a secondary index is reloaded the secondary key sequence numbers are reassigned. Therefore, the history of which records were written to the file file first is lost.

Note: If your application is dependent on the secondary key sequence number history, you would not want to rebuild your secondary index because the sequence numbers are reset.

RECOVERY WITHOUT BACKUP

If you do not use the backup procedures as described previously under "File Backup and Recovery" on page IG-94, and you encounter a problem with your file, you still may be able to recreate your file. However, the status of requests that were in process at the time of the problem is uncertain.

To recreate your file, follow the steps in "Reorganizing an Indexed File" to reorganize your file. After recreating the file, verify the status of the requests that were in process when the problem occurred.

REORGANIZING AN INDEXED FILE

An indexed file must be reorganized when a record cannot be inserted because of lack of space. This condition does not necessarily mean that there is no more space in the file; it means that there is no space in the area where the record would have been placed. Therefore, you may be able to reorganize without increasing the size of the file. Perform the following steps to reorganize a file:

1. Ensure that all outstanding requests against the file have been completed; issue a DISCONN for every current IACB.

2. Use the set parms (SE) or define (DF) commands of the $IAMUT1 utility to define a new indexed file. Estimate the number of base records and the amount and mix of free space in order to minimize the need for future reorganizations. See "Chapter 3. Defining Primary Index Files" for guidelines for making these estimates.

   You can use Option 3 of the SE command to define the new file like the original indexed file.

3. Use the reorganize command (RO) of the $IAMUT1 utility to load the new indexed file from the indexed file to be reorganized.

   Alternatively, you can use the unload command (UN) of the $IAMUT1 utility to transfer the data from an indexed file to a sequential file, then use the load command (LO) to load it back into the indexed file.

4. Use the $DISKUT1 utility to delete the old file and rename the new file.

REORGANIZING A SECONDARY INDEX: Reorganizing a secondary index does not reset the secondary key sequence numbers during the reorganization. The records are placed in another Indexed Access Method file without any modification within the individual records. The secondary key sequence numbers will be reset however, when the index is loaded.

DUMPING AN INDEXED FILE

To produce a hexadecimal dump of an indexed file, use the DP command of the $DISKUT2 utility. The dump includes control information, index blocks, and data blocks. For information on the $DISKUT2 utility, refer to the <u>Operator's Reference</u>. The <u>Indexed Access Method Version 2 Internal Design</u> contains information on dumping an indexed file.

DELETING AN INDEXED FILE

Delete an indexed file the same way you delete any other file. From a terminal, use the DE command of the $DISKUT1 utility; from a program, use the $DISKUT3 data management utility. (Refer to the <u>Operator's Reference</u> for a description of $DISKUT1, and to the <u>System Guide</u> for a description of $DISKUT3).

VERIFYING AN INDEXED FILE

$VERIFY helps you check the validity of an indexed file and prints control block and free space information about the file on $SYSPRTR.

With $VERIFY you can:

- Verify that all pointers in an indexed file are valid and that the records are in ascending sequence by key.
- Print a formatted File Control Block (FCB) listing, including the FCB Extension block. The FCB Extension block contains the original file definition parameters.
- Print a report showing the distribution of free space in your file.
- Verify secondary files against primary files.

For details on using $VERIFY, see "Chapter 9. The $VERIFY Utility" on page IG-189.

CHAPTER 7. CODING THE INDEXED ACCESS METHOD REQUESTS

This chapter describes the syntax used to code Event Driven Language requests for the Indexed Access Method.

The information in this chapter is intended for use as a reference when coding EDL application programs that use the Indexed Access Method. For information on coding Indexed Access Method applications in other languages, refer to the appropriate language manual.

Included for each request is a description of the purpose of the request, the detailed coding syntax, a description of each parameter, and all of the return codes associated with using these requests.

At the end of this chapter is a summary of the syntax of the EDL CALL instructions used to invoke the functions provided by the Indexed Access Method.

For a complete example of using the Indexed Access Method requests, refer to "Appendix C. Coding Examples" on page IG-233.

REQUEST FUNCTIONS OVERVIEW

This section provides an overview of the Indexed Access Method requests and how to code them. The Indexed Access Method callable requests are:

Request	Description
DELETE	Deletes a single record, identified by its key, from the file. Use DELETE to delete a record; the record cannot have been retrieved for update.
DISCONN	Disconnects an IACB from an indexed file, thereby releasing any locks held by that IACB; writes out all buffers associated with the file; and releases the storage used by the IACB.
ENDSEQ	Terminates sequential processing.
EXTRACT	Provides information about the file from the File Control Block, File Control Block Extension, and data paging statistics.
GET	Directly retrieves a single record from the file. If you specify the update mode, the record is locked (made unavailable to other requests) and held for possible modification or deletion. Use GET to retrieve a single record from the file.
GETSEQ	Sequentially retrieves a single record from the file. If you specify update mode, the block containing the record is locked

	(made unavailable to other requests) and held for possible modification or deletion. Use GETSEQ when you are performing sequential operations.
LOAD	Builds an Indexed Access Control Block (IACB) and connects it to an indexed file. You can then use the IACB to issue LOAD requests to that file to load records.
PROCESS	Builds an Indexed Access Control Block (IACB) and connects it to an indexed file. You can then use the IACB to issue requests to that file to read, update, insert, and delete records. A program can issue multiple PROCESS functions to obtain more than one IACB for the same file, enabling the file to be accessed by several requests concurrently within the same program.
PUT	Loads or inserts a new record depending on whether the file was opened with the LOAD or PROCESS request. Use PUT when you are adding records to a file.
PUTDE	Deletes a record that is being held for update. Use PUTDE to delete a record that has been retrieved in update mode.
PUTUP	Replaces a record that is being held for update. Use PUTUP to modify a record.
RELEASE	Releases a record that is being held for update. Use RELEASE when a record that was retrieved for update is not changed.

CODING INDEXED ACCESS METHOD REQUESTS

All Indexed Access Method services are requested by using the CALL instruction. Parameters on the CALL instructions can have the following forms:

NAME: passes the value of the variable with the label 'NAME'

(NAME): passes the address of the variable 'NAME' or the value of a symbol defined using an EQU statement

For additional information, refer to the description of the CALL instruction in the Language Reference.

General Statement Format

The general form of all Indexed Access Method calls is as follows:

```
CALL   IAM,(func),iacb,(parm3),(parm4),(parm5)
```

The request type is determined by the operand 'func'. In addition to the function request, you will notice that some functions allow a suffix of C, R, or CR. The C means perform the function requested conditionally. The condition is that the function is to be executed only if the record, the block containing the record, or the buffer containing the record is not locked. If any of those three items are locked for the record being requested, control is to be returned to the requesting program immediately. A return code is set to indicate that a lock was encountered. A conditional request can still wait on a resource if it is during the process of updating an index for a delete or insert.

The appended character, R, means return the record and the relative block number (RBN) of the record being requested. Again this can be a conditional request by preceding the letter R with the letter C. The combination CR, indicates that the record and RBN is to be returned conditionally; return the record and RBN only if the record, block, or buffer is not locked by another request.

If the RBN is requested and the record, block, and buffer are free, the RBN is returned as a 4-byte value. The 4-byte RBN value is returned at the end of the retrieved record. Therefore, when using the suffix R, ensure that your buffer is large enough to accommodate the record length, plus the 4-byte RBN value.

The RBN can be used if you are building or maintaining your own secondary index. However, because records in an indexed file are subject to being moved to different locations (RBNs) due to insert and delete activity, the RBN is not guaranteed to remain accurate if insert and delete activity to the primary index file occur.

The option of C, R, or CR is indicated in the boxed instructions with a vertical bar (|). The presence of this bar indicates that a choice must be made. Only one of the requests can be used in any one statement. For example, PUT|PUTC, you must choose one or the other when coding the request.

Depending on the type of function the remaining parameters may or may not be required. The symbols used for func and parm5 are provided by EQU statements in the IAMEQU copy code module and are coded as shown in the syntax descriptions. These symbols are treated as addresses; therefore the MOVEA instruction should be used if it is necessary to move them into a parameter list. Since these symbols are equated to constants, they may also be manipulated using other instructions by prefixing them with a plus (+) sign. Use the COPY statement to include IAMEQU in your program.

Note: You can not use the software registers (#1 and #2) on Indexed Access Method calls.

Using Program Variables

If you use variables for parameters parm3, parm4, and parm5 (that is, you code them without parentheses or a plus sign), they are set to zero by the Indexed Access Method before returning. Those parameters must be reinitialized before executing the CALL instruction again.

Link-edit Considerations

Programs which call the Indexed Access Method must be processed by $EDXLINK to include the subroutine module IAM. IAMEQU has an EXTRN statement for IAM. Refer to the chapter on "$EDXLINK" in the <u>Program Preparation Guide</u> for information on how to perform the link-edit process.

Return Codes

All Indexed Access Method requests pass a return code reflecting a condition that prevailed when the request completed. This code is passed in the task code word (referred to by task name) of the TCB associated with the requesting task. These return codes fall into three categories:

```
-1       = Successful completion
Positive = Error
Negative = Warning (other than -1)
```

Note: Return codes 1, 7, 8, and 22 are positive value return codes but they do not cause the error exit routine to be entered, even when ERREXIT is coded. Also the negative (warning) return codes do not cause error exits. For details on coding ERREXIT, see "LOAD - Open

File for Record Loading" on page IG-118, or "PROCESS - Open File" on page IG-122.

The return codes associated with each request are included with the description of the request.

The Indexed Access Method also has the capability of logging errors in the system error log. Automatic updates for secondary indexes could encounter several errors within one request. These errors will be logged in the system error log if $LOG is active. This may provide additional information when analyzing errors. For a description of $LOG, refer to the System Guide.

CALL FUNCTION DESCRIPTIONS

The Indexed Access Method CALL functions are described on the following pages and are arranged in alphabetic order.

DELETE - DELETE RECORD

The DELETE request deletes a specific record from the file. The record to be deleted is identified by its key. The deletion makes space available for a future insert. The file must be opened in the PROCESS mode.

The DELETEC request deletes a specific record from the file only if the record, block, or buffer is not locked.

Syntax:

```
label      CALL      IAM,(DELETE|DELETEC),iacb,(key)

Required:  all
Defaults:  none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(key)	The label of your key area containing the full key identifying the record to be deleted.

DELETE Return Codes

Code	Condition
-1	Successful
-85	Record not found
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an incorrect index block type was found, recreate the file
22	Address supplied by your program is not a valid IACB
76	DSOPEN error occurred - The system error field in the OPEN table contains the DSOPEN errors: 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
80	Write error - FCB. See system return code
100	Read error - check system return code
101	Write error - check system return code
230	Directory read error form $IAMDIR - check system return code
242	Secondary index is out of sync with primary file. Must rebuild file to get back in sync
244	Error in opening auto-update on secondary modification request
245	Auto update PUTDE to a secondary failed, Auto-update processing continues
247	During auto-update processing a GETSEQ to a secondary failed, auto-update processing continues

DELETE Example

The following example deletes the record whose key is 'KEY0001' from the file. The file is identified by the field named 'FILE1'.

```
        CALL   IAM,(DELETE),FILE1,(KEY)
          .
          .
          .
FILE1   DATA  F'0'    IACB ADDRESS FROM PROCESS
KEY     TEXT  'KEY0001',LENGTH=7
```

DISCONN - CLOSE FILE

The DISCONN request disconnects an IACB from an indexed file and releases the storage used for the IACB. It releases any locks held by that IACB and writes out any modified block from the file that are being held in the system buffer. Other users connected to this file are not affected.

Syntax:

```
label       CALL      IAM,(DISCONN),iacb


Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS or LOAD.

DISCONN Return Codes

Code	Condition
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
12	Data set shut down due to error; see version 2 guide, error recovery
13	Module not included in load module
22	Address supplied by your program is not a valid IACB
100	Read error - check system return code
101	Write error - check system return code
110	Write error, file closed

DISCONN Example

The following example closes the file identified by the field named 'FILE1'.

```
        CALL   IAM,(DISCONN),FILE1
         .
         .
         .
FILE1   DATA  F'0'    IACB ADDRESS FROM PROCESS
```

ENDSEQ - END SEQUENTIAL PROCESSING

The ENDSEQ request ends sequential processing, during which a block is locked and fixed in the system buffer. Sequential processing is normally terminated by an end-of-data condition. The ENDSEQ request is useful for freeing the locked block when the sequence need not be completed. ENDSEQ is valid only during sequential processing.

> **Note:** After sequential processing has been terminated, it can be restarted again anywhere in the file.

Syntax:

```
label       CALL      IAM,(ENDSEQ),iacb


Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.

ENDSEQ Return Codes

Code	Condition
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
22	Address supplied by your program is not a valid IACB

ENDSEQ Example

The following example ends sequential processing for the file identified by the field named 'FILE1'.

```
         CALL   IAM,(ENDSEQ),FILE1
          .
          .
          .
FILE1    DATA   F'0'     IACB ADDRESS FROM PROCESS
```

EXTRACT - GET FILE INFORMATION

The EXTRACT function returns information to the calling program. On a specific call, it performs one of the following:

- Returns information from a File Control Block (FCB). The FCB contains such things as the block size, key length, and data set and volume names of the indexed file. The FCBEQU copy code module contains a set of equates to map the File Control Block.

  An EXTRACT request issued for a secondary file retruns the primary FCB with the secondary key size and key position for the secondary index. If you want the FCB of the secondary file, you must open the secondary index with the independent option then the secondary index FCB will be returned. The FCB extension returned is always the FCB extension for the secondary index.

- Returns information from a File Control Block Extension. The FCB Extension contains the parameters used to define the file. The FCBEQU copy code module contains a set of equates to map the FCB Extension.

- Returns data paging statistics. These can be used to calculate page "hit" ratios.

- Returns data paging statistics, then resets them to begin accumulating new statistics.

Syntax

```
label   CALL   IAM,(EXTRACT),iacb,(buff),(size),(type)

Required: iacb (only if type is FCBNRM or FCBEXT)
          buff
Defaults: size = Full FCB
          type = FCBNRM
```

Operands	Description
iacb	The label of a word containing the IACB address returned by process or load. Required only if type=FCBNRM or FCBEXT; otherwise ignored.
(buff)	The label of the user area into which the data is returned.
	If type=FCBNRM or FCBEXT, the File Control Block is returned in this area. The area must be large enough to contain the requested portion of the FCB. Use the COPY statement to include FCBEQU in your program so that the FCB and FCB Extension fields can be referenced by symbolic names.

If type=PAGST or PAGSTR, the paging statistics are returned in this area. In this case, the size parameter is ignored, and this area must be 16 bytes in length to accommodate the statistics. The paging statistics are returned in four double-word fields:

1. Write Miss Count
2. Write Hit Count
3. Read Miss Count
4. Read Hit Count

(size) Used only if type=FCBNRM or FCBEXT; otherwise ignored. The number of bytes of the FCB or FCB Extension to be copied. The size of the FCB is the value of the symbol FCBSIZE in the FCBEQU equate table. The size of the FCB Extension is the value of the symbol FCBXSIZ in the FCBEQU equate table. Either of these symbols can be coded as the size parameter.

(type) Type of data to be returned. The following are defined:

FCBNRM Extract the FCB.

FCBEXT Extract the FCB Extension.

PAGST Returns data paging statistics to the buffer. It always returns 16 bytes.

PAGSTR Same as PAGST, except the data paging statistics are reset to zero after being copied to the buffer. This allows a new set of statistics to be accumulated.

EXTRACT Return Codes

Code	Condition
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
22	Address supplied by your program is not a valid IACB
100	Read error - check system return code
120	Invalid extract type
122	File does not contain FCB extension
123	Cannot extract paging statistics. Data paging not active

EXTRACT Examples

The following example retrieves the current paging statistics and places then in the four double words provided.

```
         CALL   IAM,(EXTRACT),0,(WRMIS),0,(PAGST)
          .
          .
          .
WRMIS    DATA  D'0'       WRITE MISS COUNT
WRHIT    DATA  D'0'       WRITE HIT COUNT
RDMIS    DATA  D'0'       READ MISS COUNT
RDHIT    DATA  D'0'       READ HIT COUNT
```

The following example gets the attributes of the file identified by the field named FILE1 from the FCB and places them in an area called WORK.

```
         CALL   IAM,(EXTRACT),FILE1,(WORK),(FCBSIZE)
.         .
          .
          .
FILE     DATA  D'0'       IACB ADDRESS FROM PROCESS
WORK     DATA  256F'0'    FCB COPY AREA
         COPY  FCBEQU     FCB EQUATES
```

GET - GET RECORD

The GET request retrieves a single record from the indexed file and places the record in a user area. The file must have been opened using the PROCESS request before the GET request is issued.

The requested record is located by key. The search may be modified by a key relation (krel) or a key length (klen). The first record in the file that satisfies the key condition is the one that is retrieved.

Retrieve for update can be specified if the requested record is intended for possible modification or deletion. The record is locked and remains unavailable to any other requests until the update is completed by a PUTUP, PUTDE or by a RELEASE. The record is also released if an error occurs or processing is ended with a DISCONN.

During an update, you must not change the primary key field in the record or the field addressed by the key parameter. The Indexed Access Method checks for and prohibits primary key modification.

The GETC request retrieves a single record from the indexed file and places the record in a user area only if the record, block, or buffer is not locked.

The GETR request retrieves the RBN of a specified record from the indexed file and places the record and RBN in a user area.

The GETCR request retrieves the RBN of a specified record from the indexed file and places the record and RBN in a user area only if the record, block, or buffer is not locked.

Syntax:

```
label      CALL   IAM,(GET|GETC|GETR|GETCR),iacb,(buff),(key),
                        (mode/krel)

Required:  iacb,buff,key
Defaults:  mode/krel=EQ
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The label of the user area into which the requested record is placed. When the RBN is requested, the RBN is returned at the end of the record. The user buffer must be four bytes longer than the record length to accommodate the RBN.

(key) The label of your key area containing the key identifying the record to be retrieved and preceded by the lengths of the key and area. This area has the standard TEXT format and may be declared using the TEXT statement. If you do not use the TEXT statement for this field, you must code it in the same format as the TEXT statement generates.

The TEXT statement format is as follows:

Offset	Field
key - 2	LENGTH (1 byte)
key - 1	KLEN (1 byte)
key	Key area ("LENGTH" bytes)

length	The length of the key area. It must be equal to or greater than the full key length for the file in use.
klen	The actual length of the key in the key area to be used as the search argument for the operation. It must be less than or equal to the full length of the keys in the file in use. If klen is 0, the full key length is assumed. A generic key search is performed when klen is less than the full key size. The first n bytes (as specified by klen) of the key area are matched against the first n bytes of the keys in the file. The first matching key determines the record to be accessed. The full key of the record is returned in the key area.
key area	The area containing the key to be used as a search argument. If you are using a generic key, after a successful GET request this area contains the full key of the record accessed.

(mode/krel) Retrieval type and key relational operator to be used. The following are defined:

EQ	Retrieve only key equal
GT	Retrieve only key greater than
GE	Retrieve only key greater than or equal
UPEQ	Retrieve for update key equal
UPGT	Retrieve for update key greater than
UPGE	Retrieve for update key greater than or equal

GET Return Codes

Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-80	End of data
-58	Record not found
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
22	Address supplied by your program is not a valid IACB
100	Read error - check system return code
101	Write error - check system return code
200	Error occurred while accessing the primary file
242	Secondary index is out of sync with primary file.
247	During auto-update processing a GETSEQ to a secondary failed, auto-update processing continues.
248	I/O error on primary file during a secondary request.
249	GET UPDATE error occurred trying to update a bad RBN.

GET Example

The following example gets a record whose key is 'JONES'. The file records are 80 bytes in length and the key length is 20 bytes. The record is returned in the area named RECORD, and because this is a GETR request, the RBN is also returned in the area named RBN, which must follow immediately after the record area.

```
         CALL   IAM,(GETR),FILE1,(RECORD),(KEY)
          .
          .
          .
FILE1    DATA  F'0'                    IACB ADDRESS FROM PROCESS
KEY      TEXT  'JONES',LENGTH=20       RECORD KEY
RECORD   DATA  128F'0'                 RECORD AREA
RBN      DATA  D'0'                    RBN
```

GETSEQ - GET RECORD (SEQUENTIAL MODE)

The GETSEQ request retrieves a single record from the indexed file and places the record in a user area (buff). The file must be opened in the PROCESS mode.

The first GETSEQ of a sequence is performed like a GET; the first record in the file that satisfies the key condition is the one that is retrieved. If key is zero, the first record in the file is retrieved. Subsequent requests in the sequence locate the next sequential record in the file and the key parameter is ignored if specified. The sequence is terminated by an end-of-data condition, by an ENDSEQ, by a DISCONN, or by an error. During the sequence, direct-access requests are invalid.

Retrieval for update can be specified if the requested record is intended for possible modification or deletion. If update is used the record is locked and remains unavailable to any other requests until the update is completed by a PUTUP, PUTDE or RELEASE. The record is also released by ending the sequence with an ENDSEQ or by ending processing with a DISCONN or by an error.

During an update, the user must not change the primary key field in the record or the field addressed by the primary key parameter. The Indexed Access Method checks for and prohibits key modification.

The GETSEQC request retrieves a single record from the indexed file and places the record in a user area only if the record, block, or buffer is not locked. The file must be opened in the PROCESS mode.

The GETSEQCR request retrieves the RBN of the specified record from the indexed file and places the record in a user area only if the record, block, or buffer is not locked. The file must be opened in the PROCESS mode.

Syntax:

```
label       CALL    IAM,(GETSEQ|GETSEQC|GETSEQR|GETSEQCR),iacb,
                    (buff),(key),(mode/krel)

Required:   iacb,buff,key
Defaults:   mode/krel=EQ
```

Operand	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The label of the user area into which the requested record is placed. When the RBN is requested, the RBN is returned at the

end of the record. The user buffer must be four bytes longer than the record length to accommodate the RBN.

(key) The label of the user key area containing the key identifying the record to be retrieved and preceded by the lengths of the key and area. If the first record of the file is to be retrieved, this field as specified should be 0. The key field, if specified, has the standard TEXT format and may be declared using the TEXT statement. If you do not use the TEXT statement for this field, you must code it in the same format as the TEXT statement generates.

The TEXT statement format is as follows:

Offset	Field
key - 2	LENGTH (1 byte)
key - 1	KLEN (1 byte)
key	Key area ("LENGTH" bytes)

length	The length of the key area. It must be equal to or greater than the full key length for the file in use.
klen	The actual length of the key in the key area to be used as the search argument for the operation. It must be less than or equal to the full length of the keys in the file in use. If klen is 0, the full key length is assumed. A generic key search is performed when klen is less than the full key size. The first n bytes (as specified by klen) of the key area are matched against the first n bytes of the keys in the file. The first matching key determines the record to be accessed. The full key of the record is returned in the key area.
key area	The area containing the key to be used as a search argument. If you are using a generic key, after the first successful GETSEQ request this area contains the full key of the record accessed.

(mode/krel) Retrieval type and key relational operator to be used. The following are defined:

EQ	Retrieve only key equal
GT	Retrieve only key greater than
GE	Retrieve only key greater than or equal

UPEQ	Retrieve for update key equal
UPGT	Retrieve for update key greater than
UPGE	Retrieve for update key greater than or equal

After the first GETSEQ of a sequence only the retrieval type is meaningful. The keys are not checked for equal or greater than relationship.

GETSEQ Return Codes

Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-80	End of data
-58	Record not found
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
22	Address supplied by your program is not a valid IACB
100	Read error - check system return code
101	Write error - check system return code
200	Error occurred while accessing the primary file
242	Secondary index is out of sync with primary file.
248	I/O error on primary file during a secondary request.
249	GET UPDATE error occurred trying to update a bad RBN.

GETSEQ Example

The following example gets the record whose key is 'KEY0001' and places it in an area called 'BUFFER'. The file is identified by the field named 'FILE1'. Subsequent GETSEQ requests result in the next sequential record being returned.

```
         CALL   IAM,(GETSEQ),FILE1,(BUFFER),(KEY)
          .
          .
          .
FILE1    DATA  F'0'            IACB ADDRESS FROM PROCESS
BUFFER   DATA  256F'0'         I/O BUFFER
KEY      TEXT  'KEY0001',LENGTH=7  RECORD KEY
```

LOAD - OPEN FILE FOR RECORD LOADING

The LOAD request builds an indexed access control block (IACB) associated with the file specified by the DSCB parameter. The address returned in the iacb variable is the address used to connect requests under this LOAD to this file.

To access the file by primary key, specify the primary file name as the DSCB parameter. On all subsequent requests, specify a primary key.

To access the file by secondary key, specify the secondary file name as the DSCB parameter. On all subsequent requests, specify a secondary key. The Indexed Access Method automatically opens the primary file when you specify a secondary file.

> **Note:** The directory must be set up to reflect the relationship among the primary file and any secondary files.

LOAD opens the file for loading base records; the only acceptable processing requests in this mode are PUT, EXTRACT and DISCONN. Only one user of a file can use the LOAD function at one time.

If an error exit is specified, the error exit routine is executed whenever any Indexed Access Method request under this LOAD terminates with a positive return code.

> **Note:** Return codes 1, 7, 8, and 22 are positive value return codes but they do not cause the error exit routine to be entered, even when ERREXIT is coded. Also the negative (warning) return codes do not cause error exits. For details on coding ERREXIT, see "LOAD - Open File for Record Loading," or "PROCESS - Open File" on page IG-122.

Syntax:

```
label       CALL      IAM,(LOAD),iacb,(dscb),(opentab),(mode)

Required:   iacb,dscb,opentab
Defaults:   mode=(SHARE)
```

Operands	Description
iacb	The label of a 1-word variable into which the address of the indexed access control block (IACB) is returned.
(dscb)	The name of a valid DSCB. This name is DSn, where n is a number from 1 - 9, corresponding to a file defined by the PROGRAM statement. It can also be a name supplied by a DSCB statement. The CALL statement specifying LOAD causes the Indexed Access Method to open the index file in load mode.

(opentab) The label of a 3 word open table. The open table contains information used during this LOAD. The format of this table is as follows:

Offset	Field
0	SYSRTCD
2	ERREXIT
4	(0) reserved

<u>Field</u>	<u>Description</u>
SYSRTCD	A 1-word variable in which the return code from LOAD and from any system function (such as READ and WRITE) is placed when requested under this LOAD by the Indexed Access Method.
ERREXIT	Your error exit routine address. If this address is zero, the error exit will not be taken. Note that error exits handle only positive return codes. **Note:** Return codes 1, 7, 8, and 22 are positive return codes which do not cause the error exit routine to be entered, even if ERREXIT is coded.
RESERVED	Must be 0 for LOAD requests.

(mode) Specifies shared or exclusive use of the file.

SHARE Allows shared read/write access by PROCESS requests.

ISHARE Allows shared read/write access by PROCESS requests with the independent processing flag on. No automatic update is performed on associated secondary indexes even if the auto-update flag is on in the directory for those secondary indexes.

For a secondary index, the index is opened as an independent file and the records returned are secondary index records, not user data records.

EXCLUSV You can access the file in exclusive mode (EXCLUSV) only if there are no outstanding PROCESS or LOAD requests. No other user can access the file while exclusive use is in effect.

IEXCLUSV You can access the file only if there are no outstanding PROCESS or LOAD requests. No other user can access the file while independent exclusive (IEXCLUSV) use is in effect. The I prefix on EXCLUSV mode prevents any automatic update functions on any associated secondary indexes, even if the auto-update flag is on in the directory entry for those associated secondary indexes.

For a secondary index, the index is opened as an independent file and the records returned are secondary index records, not user data records.

LOAD Return Codes

Code	Condition
-79	Warning - File was opened and not closed during the last session. Normal processing continues
-75	Warning - File has either not been formatted, or the invalid indicator is on in the directory for that file
-57	Data set has been loaded
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an index block, an incorrect block type was found.
17	IAM is inactive - not enough storage available. Use $IAMUT1 BF command to readjust storage size.
23	Insufficient number of IACBs, use BF command of $IAMUT1 to allocate more
50	File opened exclusive
51	Data set already opened in load mode
52	File in use, cannot open exclusive
54	$IAM buffer too small to process a file with this block size Use the BF command of $IAMUT1 to increase the buffer size
55	Insufficient FCBs
56	Read error - FCB. Refer to system return code

Code	Condition
76	DSOPEN error occurred - The system error field in the open table contains the DSOPEN errors 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
77	Record save area not large enough - use $IAMUT1 BF command to set maximum record size for secondary index processing
78	Attempted to open a secondary file for LOAD, file is not opened independently
230	Directory READ error from $IAMDIR. Check system return code
234	Directory error - DSNAME,VOL not found in $IAMDIR
243	Primary file failed to open on a secondary OPEN request

LOAD Example

The following example opens the file identified by 'DS3' for record loading in exclusive mode. The field named 'IACB' is set to the address of the IACB for this open. Subsequent requests use this field to refer to this file. The system return code is placed in the field named 'OPEN'. An error opening the file results in the routine named 'ERROR' being executed.

```
      CALL   IAM,(LOAD),IACB,(DS3),(OPEN),(EXCLUSV)
        .
        .
        .
IACB  DATA   F'0'
OPEN  DATA   F'0'         RETURN CODES
      DATA   A'ERROR'     ERROR EXIT ROUTINE ADDRESS
      DATA   F'0'         NOT USED
```

PROCESS - OPEN FILE

The PROCESS request builds an indexed access control block (IACB) associated with the file specified by the DSCB parameter. The address returned in the IACB variable is the address used to connect requests under this PROCESS to this file.

To access the file by primary key, specify the primary file name as the DSCB parameter. On all subsequent requests, specify a primary key.

To access the file by secondary key, specify the secondary file name as the DSCB parameter. On all subsequent requests, specify a secondary key. The Indexed Access Method automatically opens the primary file when you specify a secondary file.

> **Note:** The directory must be set up to reflect the relationship between the primary file and any secondary files.

PROCESS opens the file for retrievals, updates, insertions, and deletions. Multiple users can PROCESS the same file. However, only one user at a time can use the LOAD function for a given file.

If ERREXIT is specified, the error exit routine is executed whenever any Indexed Access Method request under this PROCESS terminates with a positive return code.

> **Note:** Return codes 1, 7, 8, and 22 are positive value return codes but they do not cause the error exit routine to be entered, even when ERREXIT is coded. Also the negative (warning) return codes do not cause error exits. For details on coding ERREXIT, see "LOAD - Open File for Record Loading" on page IG-118, or "PROCESS - Open File."

If EODEXIT is specified, the end-of-data exit routine is executed whenever a GETSEQ associated with PROCESS attempts to access a record after the last record in the file.

Syntax:

```
label       CALL  IAM,(PROCESS),iacb,(dscb),(opentab),(mode)

Required:   iacb,dscb,opentab
Defaults:   mode=(SHARE)
```

Operands	Description
iacb	The label of a 1-word variable into which the address of the indexed access control block (IACB) is returned.
(dscb)	The name of a valid DSCB. This name is DSn, where n is a number from 1 - 9, corresponding to a file defined by the PROGRAM

statement. It can also be a name supplied by a DSCB statement. The CALL statement specifying PROCESS, causes the Indexed Access Method to open the index file in process mode.

(opentab) The label of a 3 word open table. The open table contains information used during this PROCESS. The format of this table is as follows:

Offset	Field
0	SYSRTCD
2	ERREXIT
4	EODEXIT

Field	Description
SYSRTCD	A 1-word variable in which the return code from PROCESS and from any system function (such as READ and WRITE) is place when requested under this PROCESS by the Indexed Access Method.
ERREXIT	Your error exit routine address. If this address is 0, the error exit will not be used. Note that error exits handle only positive return codes.
EODEXIT	Your end-of-data exit routine address. If this address is 0, the end-of-data exit will not be used.

(mode) Specifies shared or exclusive access to the file.

SHARE — Allows shared read/write access by multiple PROCESS or LOAD requests.

ISHARE — Allows shared read/write access by PROCESS requests with the independent processing flag on. No automatic update is performed on associated secondary indexes even if the auto-update flag is on in the directory for those secondary indexes.

For a secondary index, the index is opened as an independent file and the records returned are secondary index records, not user data records.

EXCLUSV — The user can access the file only if there are no outstanding PROCESS or LOAD requests. No other user can access the file while EXCLUSV (exclusive access) is in effect.

IEXCLUSV — You can access the file only if there are no outstanding PROCESS or LOAD requests. No other user can access the file while independent exclusive (IEXCLUSV) use is in effect. The I prefix on EXCLUSV mode prevents any automatic update functions on any associated secondary indexes, even if the auto-update flag is on in the directory entry for those associated secondary indexes.

For a secondary index, the index is opened as an independent file and the records returned are secondary index records, not user data records.

PROCESS Return Codes

Code	Condition
-79	Warning - File was opened and not closed during the last session. Normal processing continues
-75	Warning - File has either not been formatted, or the invalid indicator is on in the directory for that file
-57	Data set has been loaded
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
17	IAM is inactive - not enough storage available. Use $IAMUT1 BF command to readjust storage size.
23	Insufficient number of IACBs, use BF command of $IAMUT1 to allocate more
50	File opened exclusive
51	Data set already opened in load mode
52	File in use, cannot open exclusive
54	$IAM buffer too small to process a file with this block size Use the BF command of $IAMUT1 to increase the buffer size
55	Insufficient FCBs
56	Read error - FCB. Refer to system return code
76	DSOPEN error occurred - The system error field in the open table contains the DSOPEN errors 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
77	Record save area not large enough - use $IAMUT1 BF command to set maximum record size for secondary index processing
78	Attempted to open a secondary file for LOAD, file is not opened independently
230	Directory READ error from $IAMDIR. Check system return code
234	Directory error - DSNAME,VOL not found in $IAMDIR
243	Primary file failed to open on a secondary OPEN request

PROCESS Example

The following example opens the file identified by 'DS1' for general access in shared access mode. The field named 'IACB' is set to the address of the IACB for this open. Subsequent requests use this field to refer to this file. The system return code is placed in the field named 'OPENTAB'. An error opening the file results in the routine named 'ERROR' being executed. An end-of-data condition on a subsequent request results in the transfer of control to the code at the label 'END'.

```
        CALL    IAM,(PROCESS),IACB,(DS1),(OPENTAB),(SHARE)
         .
         .
         .
OPENTAB DATA    F'0'         RETURN CODES
        DATA    A(ERROR)     ADDRESS OF ERROR EXIT ROUTINE
        DATA    A(END)       ADDRESS OF EOD EXIT ROUTINE
IACB    DATA    F'0'
```

PUT - PUT RECORD INTO FILE

The PUT request processes the record that is in your buffer (buff) according to the way the file was opened (LOAD or PROCESS).

If the current open is for LOAD, the record must have a higher key than the highest key already in the file and only base record slots are used (refer to "Loading Base Records From An Application Program" on page IG-65 for a description of load mode). If the current open is for PROCESS, the record may have any key and is placed in key order in either a base record or in a free slot in the appropriate place in the file.

The PUTC request processes the record that is in your buffer (buff) according to the way the file was opened (LOAD or PROCESS) only if the record or file is not locked.

Syntax:

```
label       CALL      IAM,(PUT|PUTC),iacb,(buff)


Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS or LOAD.
(buff)	The label of the user area containing the record to be added to the file.

PUT Return Codes

Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an incorrect index block type was found, recreate the file
22	Address supplied by your program is not a valid IACB
60	Out of sequence or duplicate key (LOAD only)
61	End of file
62	Duplicate key found (PROCESS only)
70	No space for insert; reorganize the file
76	DSOPEN error occurred - The system error field in the OPEN table contains the DSOPEN errors: 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
90	Internal key save area temporarily in use by another request
100	Read error - check system return code
101	Write error - check system return code
230	Directory read error form $IAMDIR. Check system return code
244	Error in opening auto-update on modification request
246	Auto-update processing an INSERT to a secondary failed, auto-update processing continues
248	I/O error on primary file during a secondary request

PUT Example

The following example puts the record in the area named 'BUFFER' into the file. The file is identified by the field named 'FILE1'.

```
         CALL    IAM,(PUT),FILE1,(BUFFER)
          .
          .
          .
FILE1    DATA  F'0'          IACB ADDRESS RETURNED HERE
BUFFER   DATA  256F'0'       I/O BUFFER
```

PUTDE - DELETE PREVIOUSLY READ RECORD

The PUTDE request deletes a record from an indexed file. The record must have been previously retrieved by a GET or GETSEQ in update mode. Deleting the record creates free space in the file. The PUTDE releases the lock placed on the record by the GET or GETSEQ.

The PUTDEC request deletes a record from an indexed file only if the block or buffer is not locked.

Syntax:

```
label       CALL      IAM,(PUTDE|PUTDEC),iacb,(buff)

Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The name of the area containing the record previously retrieved by GET or GETSEQ.

PUTDE Return Codes

Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an incorrect index block was found. Recreate the file
22	Address supplied by your program is not a valid IACB
76	DSOPEN error occurred - The system error field in the OPEN table contains the DSOPEN errors: 21 - DSNAME,VOLUME not found; 22 - VOLSERV error; 23 - I/O error
85	Key was modified by user
100	Read error - check system return code
101	Write error - check system return code
230	Directory read error form $IAMDIR. Check system return code
242	Secondary index is out of sync with primary file. Must rebuild file to get back in sync.
244	Error in opening auto-update on modification request
245	Auto update PUTDE to a secondary failed, Auto-update processing continues.
247	During auto-update processing a GETSEQ to a secondary failed, auto-update processing continues.
248	I/O error on primary file during a secondary request.

PUTDE Example

The following example deletes the record in the area named 'BUFFER' from the file. The record was read with either a GET or GETSEQ request in update mode. The file is identified by the field named 'FILE1'.

```
         CALL   IAM,(PUTDE),FILE1,(BUFFER)
          .
          .
          .
FILE1    DATA  F'0'          IACB ADDRESS FROM PROCESS
BUFFER   DATA  256F'0'       I/O BUFFER
```

PUTUP - UPDATE RECORD

The PUTUP request replaces the record in the file with the record in your buffer. The record must have been retrieved by a GET or GETSEQ in update mode. You must not change the primary key field in the record or the contents of the key area in your program returned by the GET or GETSEQ request. The Indexed Access Method checks for and prohibits primary key modification. The PUTUP releases the lock placed on the record by the GET or GETSEQ.

The PUTUPC request replaces the record in the file with the record in your buffer only if the record, block, or buffer is not locked.

Syntax:

```
label       CALL      IAM,(PUTUP|PUTUPC),iacb,(buff)

Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The label of the user area containing the record to replace the one previously retrieved.

PUTUP Return Codes

Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an incorrect index block was found. Recreate the file
22	Address supplied by your program is not a valid IACB
76	DSOPEN error occurred - The system error field in the OPEN table contains the DSOPEN errors: 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
85	Key was modified by user
100	Read error - check system return code
101	Write error - check system return code
230	Directory read error form $IAMDIR. Check system return code
242	Secondary index is out of sync with primary file. Must rebuild file to get back in sync.
244	Error in opening auto-update on modification request
245	Auto update PUTDE to a secondary failed, auto-update processing continues.
246	Auto-update processing an INSERT to a secondary failed, auto-update processing continues
247	During auto-update processing a GETSEQ to a secondary failed, auto-update processing continues.
248	I/O error on primary file during a secondary request.

PUTUP Example

The following example puts the updated record in the area named 'BUFFER' back into the file. The record was read with either a GET or GETSEQ request in update mode. The file is identified by the field named 'FILE1'.

```
         CALL   IAM,(PUTUP),FILE1,(BUFFER)
          .
          .
          .
FILE1    DATA  F'0'          IACB ADDRESS FROM PROCESS
BUFFER   DATA  256F'0'       I/O BUFFER
```

RELEASE - RELEASE RECORD

The RELEASE request frees a record that has been locked by a GET or GETSEQ for update. A record lock is normally released by a PUTUP or PUTDE. The RELEASE request is useful for freeing the locked record when the update need not be completed. RELEASE is valid only when a record is locked for update.

Syntax:

```
label       CALL       IAM,(RELEASE),iacb


Required:   all
Defaults:   none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.

RELEASE Return Codes

Code	Condition
-1	Successful
7	Link module in use, syncronize use of link module with the program
8	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
22	Address supplied by your program is not a valid IACB

RELEASE Example

The following example releases the record that was read with either a GET or GETSEQ request in update mode. The file is identified by the field named 'FILE1'.

```
         CALL   IAM,(RELEASE),FILE1
           .
           .
           .
FILE1    DATA  F'0'         IACB ADDRESS FROM PROCESS
```

EDL CALL FUNCTIONS SYNTAX SUMMARY

Following is a summary of the syntax of the EDL CALL instructions used to invoke the functions provided by the Indexed Access Method.

```
label    CALL   IAM,(DELETE|DELETC),iacb,(key)
label    CALL   IAM,(DISCONN),iacb
label    CALL   IAM,(ENDSEQ),iacb
label    CALL   IAM,(EXTRACT),iacb,(buff),(size),(type)
label    CALL   IAM,(GET|GETC|GETR|GETCR),iacb,(buff),(key),(mode/krel)
label    CALL   IAM,(GETSEQ|GETSEQC|GETSEQCR|GETSEQR),iacb,(buff),
                    (key),(mode/krel)
label    CALL   IAM,(LOAD),iacb,(dscb),(opentab),(mode)
label    CALL   IAM,(PROCESS),iacb,(dscb),(opentab),(mode)
label    CALL   IAM,(PUT|PUTC),iacb,(buff)
label    CALL   IAM,(PUTDE|PUTDEC),iacb,(buff)
label    CALL   IAM,(PUTUP|PUTUPC),iacb,(buff)
label    CALL   IAM,(RELEASE),iacb
```

INDEXED ACCESS METHOD RETURN CODES SUMMARY

Return Code	Condition
-90	Request cancelled because the request was conditional and a wait on a lock or buffer would be required
-85	Record to be deleted not found
-80	End of data
-79	Warning - File was opened and not closed during the last session, normal processing continues
-75	Warning - File has either not been formatted or the invalid indicator is on in the directory for that file
-58	Record not found
-57	Data set has been loaded
-1	Successful completion
01	Invalid function specified on CALL to $IAM
07	Link module in use, syncronize use of link module with the program
08	Load error for $IAM, verify $IAM exists and enough storage is available to load it
10	Invalid request
12	Data set shut down due to error; see version 2 guide, error recovery
13	A required module is not included in $IAM
14	Invalid index block found - during processing an incorrect index block type was found, recreate the file
17	IAM is inactive - not enough storage available Use $IAMUT1 BF command to readjust storage size
22	Address supplied by your program is not a valid IACB
23	Insufficient number of IACBs, use BF command of $IAMUT1 to allocate more
50	Data set is opened for exclusive use, cannot be opened exclusively
51	Data set already opened in load mode
52	Data set is opened, cannot be opened exclusively
54	$IAM buffer too small to process a file with this block size Use the BF command of $IAMUT1 to increase the buffer size
55	Get storage error - FCB
56	READ error - FCB, refer to system return code
60	Out of sequence or duplicate key
61	End of file
62	Duplicate key found in process mode
65	Inconsistent free space parameters RSVIX/RSVBLK or FPOOL/DYN were specified without one or the other
70	No space for insert. Reorganize the file

Return Code	Condition
76	DSOPEN error occurred - The system error field in the OPEN table contains the DSOPEN errors: 21 - DSNAME,VOLUME not found 22 - VOLSERV error 23 - I/O error
77	Record save area not large enough - use $IAMUT1 BF command to set maximum record size for secondary file processing
78	Attempted to open a secondary file for LOAD, file is not opened independently
80	FCB WRITE error during DELETE processing - see system return code
85	Key field modified by user
90	Internal key save area temporarily in use by another request
100	READ error - check system return code
101	WRITE error - check system return code
110	WRITE error - data set closed
120	Invalid EXTRACT type
122	File does not contain FCB extension
123	Cannot extract paging statistics. Data paging is not active
150	Not enough storage available for data paging
200	Error occurred while accessing the primary file
230	Directory read error form $IAMDIR
231	$IAMQCB not found. Check sysgen for include of $IAMQCB
234	Directory error - DSNAME,VOL not found in $IAMDIR
242	Secondary index is out of sync with primary file. Must rebuild file to get back in sync.
243	Primary file failed to open on secondary open request
244	Primary I/O error
245	Error in opening an auto-update modification request
246	Auto-update processing an INSERT to a secondary failed, auto-update processing continues
247	During auto-update processing a GETSEQ to a secondary failed, auto-update processing continues
248	I/O error on primary file during a secondary request
249	GET UPDATE error occurred trying to update a bad RBN

Note: For return codes 243 through 249, multiple errors may have occured. Use $ILOG to display the errors.

CHAPTER 8. THE $IAMUT1 UTILITY

This chapter describes how to use the $IAMUT1 utility to build and maintain your indexed files. Each command is described, including its function, parameters, and an example of how to use it. The file definition parameters are also described.

The chapter is arranged in alphabetic order. Following is the list of commands and the location of their descriptions:

The subcommands of the directory function (DR), are listed alphabetically under the DR description. Those subcommands are:

AL - allocate/reallocate directory
EN - end directory function
DE - delete entry
IE - insert entry
LE - list entries
UE - update entry

$IAMUT1

$IAMUT1 can be invoked using the $L command, $JOBUTIL, or the Session Manager. $IAMUT1 functions use dynamic storage for work and buffer areas. The $IAMUT1 utility is shipped with sufficient dynamic storage to handle input and output block sizes of up to 512 bytes. This enables you to define an indexed file with a maximum block size of 512 bytes, and to load, unload, and reorganize indexed files with a maximum block size of 512 bytes. $IAMUT1 determines if enough dynamic storage has been provided. If sufficient storage has not been provided, $IAMUT1 displays a message. In order to handle large blocks of data, a larger dynamic storage area will have to be provided to $IAMUT1. Additional dynamic storage can be provided by one of two ways: provide the storage parameter on the $L command, or use the SS command of the $DISKUT2 utility.

The load, unload and reorganize functions use the entire dynamic storage available to minimize the number of disk I/O operations. Improved performance, therefore, can be obtained by specifying as large a dynamic area as possible.

$IAMUT1 COMMANDS

The commands available under $IAMUT1 are listed below. To display this list at your terminal, enter a question mark in response to the prompting message ENTER COMMAND (?):.

The command descriptions in this chapter are arranged in alphabetic order.

```
ENTER COMMAND (?): ?

EC - SET/RESET ECHO MODE
EF - DISPLAY EXISTING FILE CHARACTERISTICS
DR - SECONDARY INDEX DIRECTORY FUNCTIONS
EN - END THE PROGRAM

SE - SET DEFINE PARAMETERS
DF - DEFINE AN INDEXED FILE
DI - DISPLAY CURRENT SE PARAMETERS
RE - RESET CURRENT VALUES FOR DEFINE

LO - LOAD INDEXED FILE FROM SEQUENTIAL FILE
RO - REORGANIZE INDEXED FILE
UN - UNLOAD INDEXED FILE TO SEQUENTIAL FILE

PG - SELECT DATA PAGING
NP - DESELECT DATA PAGING
PP - DEFINE PAGING PARTITIONS
PS - DATA PAGING STATISTICS
BF - SET BUFFER SIZES

ENTER COMMAND (?):
```

After the commands are displayed, you are again prompted with ENTER COMMAND (?):. Respond with the command you wish to use.

BF - TAILOR THE INDEXED ACCESS METHOD BUFFERS

The BF command specifies the amount of storage that the Indexed Access Method ($IAM) is to use for buffers and control blocks and the maximum record size for any file with a secondary index.

BF prompts you for each of the following parameters by displaying the current value and accepting new settings.

BUFFER SIZE	Indicates the amount of storage (in bytes) to be used for the central buffer. Use the following formula to calculate your minimum buffer size:

```
Buffer Size = (2 x blocksize) + (28 x blocksize/256)
where: blocksize = maximum block size
```

NUMBER OF IACBs	Indicates the number of the IACBs. The maximum number of IACBs is 64. There is an IACB associated with each PROCESS or LOAD that is issued. When calculating the number of IACBs you should consider the number of concurrent users you may have at any one time.
NUMBER OF FCBs	Indicates the number of FCBs. The maximum number of FCBs is 64. There is one FCB for every file that is open. When calculating the number of FCBs you should consider the maximum number files that might be open at a given time.
MAXIMUM RECORD SIZE	Indicates the maximum record size of any file with an associated secondary index. If no files have a secondary index, this value can be zero. The actual amount of storage reserved as a result of this parameter is twice the value specified plus 8 bytes.

None of these take effect until the next time the Indexed Access Method is loaded.

BF Command Example

This example sets the central buffer size to 540 bytes, leaves the number of IACBs at 3, leaves the number of FCBs at 3, and sets the maximum record size of any file with a secondary index to 120 bytes.

```
ENTER COMMAND (?): BF
PARAMETER                DEFAULT    NEW VALUE
BUFFER SIZE                 1080  : 540
NUMBER OF IACBs                3  :
NUMBER OF FCBS                 3  :
MAXIMUM RECORD SIZE          256  : 120
VALUE(S) SET
STORAGE FOR $IAM HAS BEEN SET TO 2048
BECOMES EFFECTIVE ON NEXT LOAD OF $IAM

ENTER COMMAND (?):
```

DF - DEFINE INDEXED FILE

The DF command allocates, defines, and formats an indexed file. The DF function will optionally invoke the load or reorganize function for you. Before entering DF, you must use the SE command to set up parameters that determine the size and format of the indexed file. The DF command uses those SE parameters to optionally allocate and format the file. The DF function can be invoked at the end of the SE function.

The allocate step consists of using the file size computed during the SE step to dynamically allocate the file. If the file already exists, the size is verified to ensure that it is large enough. The define step consists of writing the file control block (FCB) and its extension to the indexed file. Finally, the optional format step initializes all records in the indexed file to provide an empty structured file.

INVOKING THE LOAD AND REORGANIZE FUNCTIONS FROM DF: You can invoke the LOAD or REORGANIZE functions directly from the DF (or SE) command. If you invoke these functions, DF does not format the file because LOAD and REORGANIZE will format the file. If you do not invoke the LOAD or REORGANIZE function, DF formats the file so you can load the file using an application program or the LO command.

Notes:

1. You can use the LOAD/REORGANIZE command later to load the file, if you do not invoke it from the DF command.

2. An application program cannot access an unformatted indexed file.

3. The prompt for the load/reorganize function occurs before the file is actually defined.

4. A secondary index file cannot be loaded with the LO command, though it can be reorganized using the RO function.

Defining the File

The define function prompt for the file to be allocated. If the file already exists, its size is checked. If the size is at least as large as needed, DF prompts you as to whether the file should be reused as follows:

```
ENTER COMMAND (?): DF
ENTER DATA SET (NAME,VOLUME) : IAMFILE,EDX003
DATA SET ALREADY EXISTS AND IS LARGE ENOUGH
DO YOU WISH TO REUSE IT (Y/N)?: Y
```

If the file exists, but it is not as large as needed, you have the option of deleting and reallocating it as shown in the following example:

```
ENTER COMMAND (?): DF
ENTER DATA SET (NAME,VOLUME) : MASTER,VOL123
DATA SET ALREADY EXISTS AND IS TOO SMALL
DELETE AND REALLOCATE (Y,N)? : Y
DELETE AND REALLOCATE COMPLETED
```

If the file does not exist, it is allocated as follows:

```
ENTER COMMAND (?): DF
ENTER DATA SET (NAME,VOLUME) : MASTER,VOL123
NEW DATA SET IS ALLOCATED
```

Using Immediate Write-Back

DF prompts you to select whether or not you want to use the immediate write-back option. Immediate write-back has the same effect on primary or secondary indexed files.

Each request to insert, delete, or update a data record, causes the affected blocks to be read into the Indexed Access Method buffer. The actual modification to the block is performed in the buffer.

If you enter N to the immediate write-back prompt, file modifications are held in the main storage buffer and not written back to the indexed file until the buffer space is needed for another block or until the file is closed. If the device where the file resides was powered off before the block was written back to the file, the modification to the file would not have been performed.

If you enter Y to the immediate write-back prompt, you are assured that the changed block is written back to the file immediately.

The prompt is as follows:

```
DO YOU WANT IMMEDIATE WRITE-BACK? Y
```

DF Command Example

The following example shows a use of the DF command to define a file named MASTER on volume VOL123. Immediate write-back is selected and the request to invoke LOAD or REORGANIZE is indicated.

```
ENTER COMMAND (?): DF
ENTER DATA SET (NAME,VOLUME) : MASTER,VOL123
NEW DATA SET IS ALLOCATED
DO YOU WANT IMMEDIATE WRITE-BACK? Y
INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION? L
DEFINE IN PROGRESS
DATA SET SIZE IN EDX RECORDS:             17
INDEXED ACCESS METHOD RETURN CODE:        -1
SYSTEM RETURN CODE:                       -1
PROCEED WITH LOAD/REORGANIZE (Y/N)
```

DI - DISPLAY PARAMETER VALUES

DI displays the current parameter values entered during the current session of $IAMUT1 SE command. The parameter values can be used to format a file using the DF command or they can be modified by reusing the SE command.

Note: You can also use the EF command to display the parameters of an existing file.

The following example shows a use of the DI command.

```
ENTER COMMAND (?): DI
CURRENT VALUES FOR SE COMMAND ARE:
FILE TYPE = PRIMARY
BASEREC          100
BLKSIZE          256
RECSIZE           80
KEYSIZE           28
KEYPOS             1
FREEREC            1
FREEBLK           10
RSVBLK          NULL
RSVIX              0
FPOOL           NULL
DELTHR          NULL
DYN             NULL
```

For a secondary file, the record size is not displayed.

DR - INVOKE SECONDARY INDEX DIRECTORY FUNCTIONS

The DR command provides access to secondary index directory functions. Those functions are made available by replying DR when $IAMUT1 requests "ENTER COMMAND (?):". You can then respond to the "ENTER DIRECTORY COMMAND (?):" with a subcommand. To obtain a list of the available subcommands, reply with a question mark (?) as follows:

```
ENTER COMMAND (?): DR

ENTER DIRECTORY COMMAND (?): ?

AL - ALLOCATE/REALLOCATE DIRECTORY
LE - LIST ENTRIES
ID - INSERT ENTRY
DE - DELETE ENTRY
UE - UPDATE ENTRY
EN - END DIRECTORY FUNCTION

ENTER DIRECTORY COMMAND (?):
```

The directory function commands are arranged in alphabetic order as DR - xx, where xx is the two letter directory subcommand. Their specific locations are listed below:

"AL - Allocate Directory" on page IG-151

"DE - Delete Directory Entry" on page IG-153

"EN - End Directory Function" on page IG-154

"IE - Insert Entry" on page IG-155

"LE - List Entries" on page IG-156

"UE - Update Directory Entry" on page IG-158

AL - ALLOCATE DIRECTORY

The AL subcommand allocates a directory for secondary indexes. If a directory already exists, this subcommand gives the option to delete and reallocate it.

Note: To use this subcommand, you must first use the DR command.

You are prompted to enter the maximum number of directory entries. Enter the number of entries you want the directory to be able to hold. Each entry describes a primary file or secondary index. The maximum number of entries defaults to 47.

The directory, $IAMDIR, is always allocated on the IPL volume.

The following example shows a use of the AL subcommand to allocate a new directory with a capacity of 10 entries:

```
ENTER DIRECTORY COMMAND (?): AL
MAX # OF DIRECTORY ENTRIES: 10
THE DIRECTORY DATA SET REQUIRES 1 EDX RECORDS, CONTINUE (Y/N/EN)? Y
DIRECTORY DATA SET ALLOCATED: $IAMDIR,EDX002
```

The next example assumes a directory already exists and allocates a new one.

```
ENTER DIRECTORY COMMAND (?): AL
DIRECTORY EXISTS, OPTIONS ARE:
BN - BUILD NEW DIRECTORY
AS - ADJUST SIZE
EN - END DIRECTORY ALLOCATE

ENTER OPTION: BN

ALL DIRECTORY ENTRIES WILL BE DELETED, CONTINUE (Y/N)? Y
MAX # OF DIRECTORY ENTRIES: 20
THE DIRECTORY DS REQUIRES      2 EDX RECORDS, CONTINUE (Y/N/EN) ? Y

DIRECTORY DATA SET ALLOCATED: $IAMDIR,EDX002
```

The following example, adjusts the size of the directory data set. All existing entries will be retained.

DR - AL

```
ENTER DIRECTORY COMMAND (?): AL
DIRECTORY EXISTS, OPTIONS ARE:
BN - BUILD NEW DIRECTORY
AS - ADJUST SIZE
EN - END DIRECTORY ALLOCATE

ENTER OPTION: AS
MAX # OF DIRECTORY ENTRIES: 1
THE DIRECTORY DS REQUIRES      1 EDX RECORDS, CONTINUE (Y/N/EN) ? Y

DIRECTORY DATA SET ALLOCATED: $IAMDIR,EDX002
```

DE - DELETE DIRECTORY ENTRY

The DE subcommand deletes an entry from the directory. If you delete a primary entry, all associated secondary index entries are also deleted.

Note: To use this subcommand, you must first use the DR command.

The following example shows the deletion of the directory entry for the file named MASTER on the volume named VOL123. MASTER is a primary index file entry which has secondary indexes associated with it.

```
ENTER DIRECTORY COMMAND (?): DE
ENTRY (DSNAME,VOLUME): MASTER,VOL123
ASSOCIATED SECONDARY ENTRIES WILL BE DELETED, CONTINUE (Y/N)?      Y
DELETE SUCCESSFUL, NUMBER OF ENTRIES DELETED:    2
```

The following example shows the deletion of the directory entry for a file named MASTER, on the volume named VOL123. MASTER is a primary index file entry which no longer has any secondary indexes associated with it.

```
ENTER DIRECTORY COMMAND (?): DE
ENTRY (DSNAME,VOLUME): MASTER
ENTRY FOR MASTER  ,EDX002 WILL BE DELETED, CONTINUE (Y/N)? Y
DELETE SUCCESSFUL, NUMBER OF ENTRIES DELETED:     1
```

EN - END DIRECTORY FUNCTION

The EN subcommand terminates the directory functions (DR) and returns to $IAMUT1 for your next command.

IE - INSERT ENTRY

The IE subcommand inserts a new entry into the secondary index directory. It is used to insert either a primary or secondary entry. However, the primary entry must be inserted before any of its secondary entries can be inserted.

For a primary entry, enter the data set name and volume of the file for which the entry is being inserted. Specify N when asked "IS THIS A SECONDARY ENTRY (Y/N)?."

For secondary entries, enter the data set name and volume of the secondary index for which the entry is being inserted and specify that it is a secondary index. You are then prompted for additional information.

Specify the name of the primary index file which the secondary index is to be associated with. You can select automatic update, which indicates that any change to a primary file is to be reflected in the secondary index. The default for automatic update is yes.

The following example inserts a directory entry for a primary index file:

```
ENTER DIRECTORY COMMAND (?): IE
ENTRY (DSNAME,VOLUME): TOMPRI,EDX002
IS THIS A SECONDARY ENTRY (Y/N)? N

DIRECTORY INSERT SUCCESSFUL
```

The following example inserts a directory entry for a secondary index named 'TOMSEC1,EDX002' which is to be associated with the primary index file 'TOMPRI,EDX002'. Automatic update is selected.

```
ENTER DIRECTORY COMMAND (?): IE
ENTRY (DSNAME,VOLUME) TOMSEC1,EDX002
IS THIS A SECONDARY ENTRY?  Y

ASSOCIATED PRIMARY ENTRY (DSNAME,VOLUME): TOMPRI,EDX002
AUTO-UPDATE (Y/N)? Y
```

Note: To use this subcommand, you must first use the DR command.

LE - LIST ENTRIES

The LE subcommand lists the contents of one or more directory entries. Specify the name of a primary indexed file to get information about that file and its secondary indexes. Specify the name of a secondary index to get information about only that secondary index. To obtain a complete list of all information in the directory, just press the Enter Key without supplying any data set name or volume.

Note: To use this subcommand, you must first use the DR command.

The following example lists the directory entries related to the primary file named 'TOMPRI' on volume 'EDX002'.

```
ENTER DIRECTORY COMMAND (?): LE
ENTRY (DSMANE,VOLUME) BLANK=ALL: TOMPRI

                PRIMARY  INDE-           AUTO
DSNAME   VOLUME DATA SET PENDENT  INVALID UPDATE

TOMPRI   EDX002 YES      NO       ****    ****
TOMSEC1  EDX002 NO       NO       YES     YES
TOMSEC2  EDX002 NO       NO       YES     NO

NUMBER OF DIRECTORY ENTRIES USED =      6
NUMBER OF AVAILABLE ENTRY SLOTS =      41
DIRECTORY LIST COMPLETED
```

The following example lists all directory entries.

```
ENTER DIRECTORY COMMAND (?): LE
ENTRY (DSNAME,VOLUME) BLANK=ALL:

                PRIMARYU  INDE-               AUTO
DSNAME   VOLUME DATA SET  PENDENT  INVALID  UPDATE

EDXIAM   EDX003 YES       NO       ****     ****
EDXIAMS1 EDX003 NO        NO       YES      YES

TOMPRI   EDX002 YES       NO       ****     ****
TOMSEC1  EDX002 NO        NO       YES      YES
TOMSEC2  EDX002 NO        NO       YES      NO

NUMBER OF DIRECTORY ENTRIES USED =      6
NUMBER OF AVAILABLE ENTRY SLOTS =     41
DIRECTORY LIST COMPLETED
```

UE - UPDATE DIRECTORY ENTRY

The UE subcommand updates an entry in the secondary index directory. You can use this command as follows:

- Specify null values for parameters to remain unchanged (press the Enter key when you are prompted for them).

- Enter new values for parameters to be modified.

  Note: You cannot change a primary entry to a secondary entry or a secondary entry to a primary entry. To do this, you must delete the old entry and insert a new one.

The following example updates a primary directory entry named 'MASTER,VOL123', changes the volume name from VOL123 to EDX002 and leaves the DSNAME MASTER as it is.

```
ENTER DIRECTORY COMMAND (?): UE

ENTRY (DSNAME,VOLUME) MASTER,VOL123

THIS IS A PRIMARY ENTRY

IN THE FOLLOWING, ENTER NEW VALUE OR,
ENTER NULL LINE TO RETAIN (PRESENT VALUE)

DSNAME (MASTER):
VOLUME (VOL123): EDX002
INDEPENDENT (N):

DIRECTORY UPDATE SUCCESSFUL
```

The following example updates a secondary directory entry named 'MASTER,VOL123', changes the VOLUME name to EDX002 and leaves the DSNAME MASTER as it is. It sets automatic update, leaves the independent processing flag as it is, and sets the invalid indicator off.

```
ENTER DIRECTORY COMMAND (?): UE

ENTRY (DSNAME,VOLUME) MASTER,VOL123

THIS IS A SECONDARY ENTRY
IN THE FOLLOWING, ENTER NEW VALUE OR,
ENTER NULL LINE TO RETAIN (PRESENT VALUE)

DSNAME (MASTER):
VOLUME (EDX123): EDX002
INDEPENDENT (N):
INVALID INDICATOR (Y): N
AUTO-UPDATE (Y): Y

DIRECTORY UPDATE ENDED
```

Note: To use this subcommand, you must first use the DR command.

EC - CONTROL ECHO MODE

EC enables you to enter or leave echo mode. When in echo mode, all $IAMUT1 input and output is logged on the $SYSPRTR device. This enables you to save information about the files you maintain using $IAMUT1. When in echo mode, all input and output is logged until either the current utility session is ended or echo mode is reset by use of the EC command. Echo mode is off when $IAMUT1 is loaded.

Note: Input and output from $DISKUT3 is not logged.

The following examples show the commands to set and reset echo mode:

```
ENTER COMMAND (?): EC
DO YOU WANT ECHO MODE? (Y/N)?: Y   (Set echo mode)
FUNCTION COMPLETED

ENTER COMMAND (?): EC
DO YOU WANT ECHO MODE? (Y/N)?: N   (Reset echo mode)
FUNCTION COMPLETED
```

EF - DISPLAY EXISTING INDEXED FILE CHARACTERISTICS

The EF command displays the file definition parameters that were used to set up the file. The information is obtained from the FCB Extension block.

EF Command Example

This example shows how to display the file parameters used to set up the file.

```
ENTER COMMAND (?): EF
EXHIBIT FUNCTION ACTIVE
ENTER DATASET (NAME,VOLUME): EDXIAM1,EDX003

FILE TYPE = PRIMARY
BASEREC           20
BLKSIZE          256
RECSIZE           80
KEYSIZE            4
KEYPOS             1
FREEREC            0
FREEBLK            0
RSVBLK          NULL
RSVIX              0
FPOOL           NULL
DELTHR          NULL
DYN               10
EXHIBIT FUNCTION COMPLETED
```

LO - LOAD INDEXED FILE

LO loads a primary indexed file from a sequential (blocked or unblocked) input file. (A secondary indexed file must be loaded by using the DF or SE command). A primary indexed file can be loaded in one of two environments. Loading an empty file is referred to as the initial load. For an indexed file that already contains some records, the LO command can be used to add records with higher keys (keys of higher value than those already in the indexed file). This is called load in **extend** environment.

Blocks are read from the sequential file with the EDL READ instruction and de-blocking is performed, if necessary. In the initial load environment, data records are formatted into Indexed Access Method blocks and written to the indexed file with the EDL WRITE instruction. Corresponding index blocks are written as required. The remainder of the indexed file is formatted if formatting was not completed during the DF function. In the extend environment, records are loaded into the indexed file using Indexed Access Method PUT requests.

The sequential input file can contain blocked or unblocked records. For a description of blocked and unblocked sequential data sets, see "Blocked and Unblocked Sequential Data Sets" on page IG-164. The records in the sequential file must be in ascending order by the data contained in the key field. If a record with a duplicate or out of sequence key is found, you are given the option to either omit the record and continue loading, or to end loading. The indexed file must have been defined by using the SE and DF commands before using the LO command.

Your response to the prompt message "ENTER INPUT BLOCKSIZE", defines to the LO command whether the input is a blocked or unblocked sequential file. A null response to the prompt "ENTER INPUT BLOCKSIZE" indicates an unblocked input file and the block size is then calculated using the input record size value, rounded up to the next 256-byte multiple value. If the actual block size value is entered as your response to this prompt, a blocked sequential input file is indicated.

The record lengths of the input and output files do not have to be the same. When the indexed file is opened, the record length is displayed on the terminal. At this point, you can specify the record length of the sequential file if it is different than that of the indexed file. If the indexed file records are longer than the sequential file records, the loaded records are left justified and filled with binary zeroes. If the indexed file records are shorter than the sequential file records, the following message appears on the terminal:

```
INPUT REC GT OUTPUT REC. TRUNCATION WILL OCCUR.
OK TO PROCEED?
```

Reply 'Y' to proceed (records will be truncated).

Reply 'N' to terminate the load function.

If the end of the input sequential file is reached, you can continue loading from another sequential file. You are asked if there is more data to load. If you reply yes (Y), you are prompted for the file and volume name of the new input sequential file to use. The load operation continues, putting the first record of the new input sequential file in the next available record slot of the indexed file.

> **Note:** The record lengths and block sizes of subsequent input files are assumed to be the same as the initial input file.

If the end of input file is reached and you do not name another input file, the load operation is complete.

> **Note:** If you are loading the indexed file from a tape file, $IAMUT1 does not close the tape file upon completion of the load. Use the $VARYOFF command to close the tape file (refer to the _Operator's Reference_ for a description of the $VARYOFF command).

The following example shows use of the LO command:

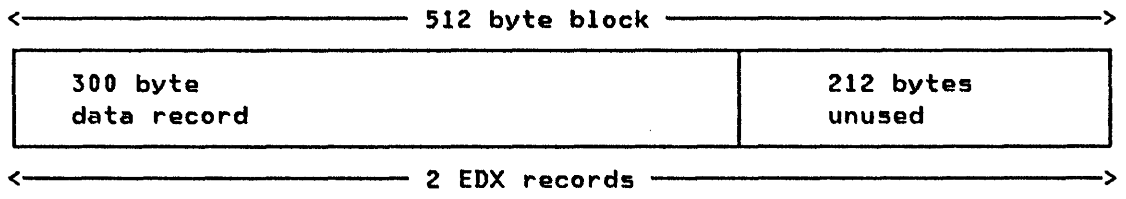
```
ENTER COMMAND (?): LO

LOAD ACTIVE
ENTER OUTPUT DATASET (NAME,VOLUME): IAMFILE,EDX003
$FSEDIT FILE RECSIZE = 128
INPUT RECORD ASSUMED TO BE     80 BYTES. OK?: Y
ENTER INPUT BLOCKSIZE (NULL = UNBLOCKED):
ENTER INPUT DATASET (NAME,VOLUME): SEQ01,EDX003
LOAD IN PROCESS

END OF INPUT DATASET
ANY MORE DATA TO BE LOADED?: N
     6 RECORDS LOADED
LOAD SUCCESSFUL
```

Blocked and Unblocked Sequential Data Sets

The LO (load) function of $IAMUT1 will accept either blocked or unblocked sequential data sets as input when loading an indexed file. The UN (unload) function will either block or unblock data as requested when unloading an indexed file to a sequential data set.

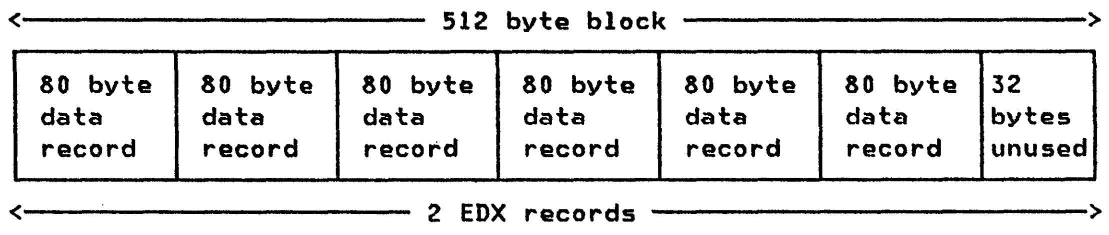
UNBLOCKED SEQUENTIAL DATA SET: An unblocked sequential data set contains one record in each block. The blocksize must be a multiple of 256 bytes. The record size must be equal or less than the block size. A block can span one or more EDX records.

The following diagram illustrates the relationship of a data record of 300 bytes to a block size of 512 bytes in an unblocked data set.



BLOCKED SEQUENTIAL DATA SET: In a blocked sequential data set a block can contain multiple logical records. The block size must be a multiple of 256 bytes. The record size must be equal to or less than the block size. A block can span one or more EDX record.

The following diagram illustrates 6 data records of 80 bytes each within a block of 512 bytes in a blocked data set.



Both the blocked and unblocked forms of sequential data sets, used by the utility, are compatible with the language processors, Sort/Merge and data sets produced by $FSEDIT. If you use the EDX edit utilities to prepare your data records for input, remember that these utilities put one 80-byte line from $FSEDIT into a 128-byte $FSEDIT record. Two of these 128-byte records are then used to form one 256-byte EDX record. When you use such a data set as sequential input for the LO (load) function, specify the record length as 128 and the block size as 256. If your indexed file is defined as having a record length of 80, you will receive the message "TRUNCATION WILL OCCUR." This is acceptable because Indexed Access Method strips off the extra bytes added by $FSEDIT.

The last block of a blocked sequential data set may not have enough records for a full block. In this case, all of the unused space in the block is set to binary zeroes.

Invoking the LOAD and REORGANIZE Functions

You can invoke the LOAD or REORGANIZE functions directly from the DF command. If you invoke these functions, DF does not format the file because LOAD and REORGANIZE will do it. If you do not invoke the LOAD or REORGANIZE function, DF formats the file so you can load the file using an application program or $IAMUT1 at a later time.

Notes:

1. You can use the LOAD/REORGANIZE command later to load the file, if you do not invoke it from the DF command.

2. An application program cannot access an unformatted indexed file.

3. The prompt for the load/reorganize function occurs before the define step.

NP - DEACTIVATE PAGING

The NP command directs that data paging be deselected the next time the Indexed Access Method is loaded.

Page area sizes are not affected by this command.

NP Command Example

This example shows how to indicate data paging is to be deselected on the next invocation of the Indexed Access Method.

```
ENTER COMMAND (?): NP
DATA PAGING MARKED AS NOT ACTIVE
BECOMES EFFECTIVE ON NEXT LOAD OF $IAM
```

PG - SELECT PAGING

The PG command directs that data paging be selected the next time the Indexed Access Method is loaded.

Page area sizes are not affected by this command.

PG Command Example

This example shows how to indicate data paging is to be selected on the next invocation of the Indexed Access Method.

```
ENTER COMMAND (?): PG
DATA PAGING MARKED AS SELECTED
BECOMES EFFECTIVE ON NEXT LOAD OF $IAM
SEE INDEXED ACCESS METHOD GUIDE CONCERNING
REMOVAL OF PAGING MODULES FROM STORAGE.

ENTER COMMAND (?):
```

PP - DEFINE PAGING PARTITIONS

The PP command defines the amount of storage in each partition that the Indexed Access Method should reserve for paging. Storage is actually used for paging only when paging is active.

PP prompts you for the size of the paging area for each partition by displaying the partition number and current paging area size for that partition. Respond with a null entry (just press the Enter key) to retain that size. Enter a new size to change the space allocation. Sizes are displayed and entered in K bytes (1K = 1024), and should be entered as even numbers (multiple of 2K). If not, they are adjusted up to the next even number. The new sizes do not take effect until the next time the Indexed Access Method is loaded with paging active.

PP Command Example

This example retains the 10K paging area size in partition 3 and increases the paging area in partition 5 from 6K to 10K.

```
ENTER COMMAND (?): PP
PARTITION  CURRENT   NEW

    1          0K  :
    2          0K  :
    3          0K  : 40
    4          0K  :
    5          0K  :
    6          0K  :
    7          0K  :
    8          0K  :
PAGE AREA SIZE(S) RESET
BECOMES EFFECTIVE ON NEXT LOAD OF $IAM
TOTAL PAGE AREA SIZE IS   40K
SEE INDEXED ACCESS METHOD GUIDE CONCERNING
REMOVAL OF PAGING MODULES FROM STORAGE.
```

Notes:

1. The letter K is optional on input, and is assumed if missing.
2. The new total page area size is 40K and becomes effective on the next LOAD of $IAM.

PS - GET PAGING STATISTICS

The PS command displays data paging information about the currently executing Indexed Access Method. It shows the total size of the paging area, and "hit" information for reads, writes and overall.

The Indexed Access Method increments a "hit" counter each time a referenced block is found in the paging area. It increments a "miss" counter each time a referenced block is not found in the paging area. The PS command displays these numbers, along with "hit percentages." Use the hit percentages to determine how efficiently the paging area is being used.

After the statistics are displayed, you have the option of resetting the counters to zero so that a new set of paging statistics can be gathered.

PS Command Example

Display the current paging statistic and reset them.

```
ENTER COMMAND (?): PS
FUNCTION       HITS     MISSES     HIT %
  READ        45678      81205      36
  WRITE        2450          0     100
  OVERALL     48128      81205      37

RESET STATISTICS (Y/N)? Y
STATISTICS RESET
```

RE - RESET PARAMETERS

RE resets the parameters set up by the SE command to their default values.

The following example shows a use of the RE command:

```
ENTER COMMAND (?): RE
PARAMETERS RESET
```

RO - REORGANIZE INDEXED FILE

RO reorganizes a primary or secondary indexed file. It unloads an indexed file filled by insert activity into an empty indexed file and reorganizes the records to provide space for additional inserts.

This command requires two existing indexed files of the same type. Both the input file and the output file must be primary indexed files, or both must be secondary index files. Records are read sequentially from the input file using the Indexed Access Method GETSEQ request. The records are loaded into the output file in a manner similar to the initial load of the LO command.

All reserved and free space is retained as free space.

Reorganizing a secondary index does not reset the secondary key sequence numbers during the reorganization, because it does not use the primary file. The records are placed in another Indexed Access Method file without any modification within the individual records.

The output indexed file must have been defined by using the SE or DF commands before using the RO command. The SE Option 3 will format an output file like the original file, or $VERIFY will show the number of records in the file so that you can set up an output file.

The record lengths of the two files need not be the same. Unloaded records are truncated or filled with binary zeroes if record lengths differ (see LO command). The key fields and key positions of the two files must be the same; however, the other file specifications (SE parameters) may differ.

INVOKING THE LOAD AND REORGANIZE FUNCTIONS FROM DF: You can invoke the LOAD or REORGANIZE functions directly from the DF command. If you invoke these functions, DF does not format the file because LOAD and REORGANIZE will do that, thus saving time. If you do not invoke the LOAD or REORGANIZE function, DF formats the file so you can load the file using an application program or the LO command.

Notes:

1. You can use the LOAD/REORGANIZE command later to load the file, if you do not invoke it from the DF command.

2. An application program cannot access an unformatted indexed file.

3. The prompt for the load/reorganize function occurs before the define step.

The following example shows use of the RO command:

```
ENTER COMMAND (?): RO

REORG ACTIVE
ENTER INPUT DATASET (NAME,VOLUME): IAMFILE,EDX003
ENTER OUTPUT DATASET (NAME,VOLUME): IAMFIL2,EDX003
REORG IN PROCESS

END OF INPUT DATASET

   100 RECORDS LOADED
REORG SUCCESSFUL

ENTER COMMAND (?): EN
```

SE - SET PARAMETERS

SE prompts you for parameters that determine the structure and size of the indexed file. An explanation of the SE command parameters follow and an example of each is included with the description.

The parameter values entered are saved by $IAMUT1. This enables you to reuse the SE command to change one or more parameters without having to reenter all of them. The current values can be displayed by the DI command.

The SE command provides three methods of setting up an indexed file.

Option 1 Significant Parameters - Enter a minimal set of SE parameters. The utility internally converts the smaller set to the complete set.

Option 2 All Parameters - Enter the complete set of SE parameters.

Option 3 Parameters from Existing Data Set - Use the set of SE parameters that were used previously to define an existing indexed file.

Note: Information which is common to all three options appears near the end of the SE description under "All Options" on page IG-182.

When you specify the SE command, you are prompted to select one of the options as shown in the following display.

```
SET FILE DEFINITION PARAMETERS
0 = EXIT
1 = SIGNIFICANT PARAMETERS
2 = ALL PARAMETERS
3 = PARAMETERS FROM EXISTING INDEXED DATASET
ENTER OPTION:
```

Option 1

Option 1 prompts for a minimal set of parameters. It issues a prompt to determine if a secondary index is being defined. If so, the secondary file name, key size, and key position are requested. If a primary file is being defined, different prompts are issued. $IAMUT1 internally converts the option 1 parameters to option 2 parameters.

When the SE option 1 is invoked for the first time, the prompts and defaults values are as follows:

```
SECONDARY INDEX (Y/N)?: N

                                          DEFAULT  NEW VALUE
RECORD SIZE                                    0: 80
KEY SIZE                                       0: 4
KEY POSITION                                   1:
BLOCKING FACTOR (RECORDS PER BLOCK)            1:
NUMBER OF BASE RECORDS                         0: 20
ESTIMATED TOTAL RECORDS                       24:
TYPE OF INSERT ACTIVITY(C=CLUSTERED,R=RANDOM)  C:
```

On subsequent invocations of the SE option 1, the defaults are taken from the parameter values since the last SE option 1 invocation. Option 1 and 3 values do not carry over to option 2.

The estimated total records value defaults to the last value, provided this value equals or exceeds the current base records. Otherwise it defaults to 1.2 times the current base records.

To set up a secondary index, enter the following:

```
SECONDARY INDEX (Y/N)?: Y
ENTER SECONDARY DATASET NAME (DS,VOL): FILE01,EDX002
SECONDARY KEY SIZE:10
SECONDARY KEY POSITION:36
```

Before you can define a secondary index, you must place an entry in the directory for the associated primary index file and the primary file must exist. The directory is searched to obtain the data set name and volume of the associated primary file which will then be used to compute the remainder of the secondary SE parameters.

Parameter Descriptions for Option 1

The attributes of the file are determined by the following SE command parameters:

RECORD SIZE: The length, in bytes, of each record in the file.

KEY SIZE: The length of the key to be used for this file. The minimum key length is 1. For primary files, the maximum key length is 254.

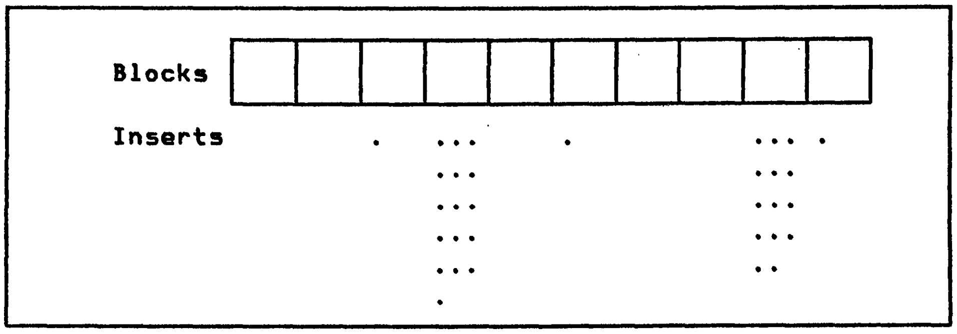
KEY POSITION: The position, in bytes, of the key within the record. The first byte of the record is position 1.

BLOCKING FACTOR (RECORDS PER BLOCK): The total number of records to be placed in an Indexed Access Method block. This value and the record size will be used to compute the actual Indexed Access Method block size, rounded up to the next 256-byte value. The rounding up action may increase the actual blocking factor.

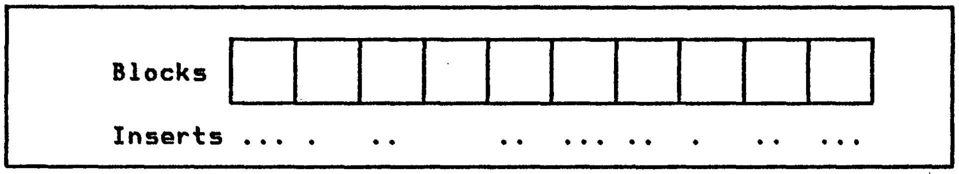
NUMBER OF BASE RECORDS: The number of indexed record slots to be set up in the indexed file. These record slots can be loaded with data records by $IAMUT1 or by a PUT request after either a LOAD or PROCESS request.

ESTIMATED TOTAL RECORDS: The total number of records you expect the indexed file to contain after insert processing activity.

TYPE OF INSERT ACTIVITY(C=CLUSTERED,R=RANDOM): Inserts are considered clustered if most of the inserts occur at only certain places in the file. The following diagram represents clustered inserts by vertically stacked bullets.



The next diagram represents random inserted records. Inserts are considered random if few or no points in the file have a concentration of activity; inserts are expected throughout the file.



SECONDARY KEY SIZE: The length, in bytes, of the secondary key within the primary record. For secondary keys the maximum key length is 250.

SECONDARY KEY POSITION: The position, in bytes, of the secondary key within the primary record.

Option 2

The following list shows the default values for parameters when the SE command is invoked the first time (all values are decimal):

BASEREC	NULL
BLKSIZE	0
RECSIZE	0
KEYSIZE	0
KEYPOS	1
FREEREC	0
FREEBLK	0
RSVBLK	NULL
RSVIX	0
FPOOL	NULL
DELTHR	NULL
DYN	NULL

On subsequent invocations of the SE command, the option 2 defaults are taken from the parameter values set according to the last SE command, regardless of the option used. If the default value is acceptable, press the enter key when prompted for the parameter. If you wish to change the value for any parameter, enter the new value in response to the prompting message. The new value becomes the new default value for the current $IAMUT1 session. The parameters for which a null can be specified are BASEREC, FREEREC, FREEBLK, RSVBLK, RSVIX, FPOOL, DELTHR, and DYN. To specify a null parameter after the original default has been modified, enter an ampersand (&) in response to the prompting message.

The following example shows a use of the SE command in establishing the size and structure of an indexed file.

```
PARAMETER    DEFAULT NEW VALUE
BASEREC         NULL :100
BLKSIZE            0 :256
RECSIZE            0 :80
KEYSIZE            0 :28
KEYPOS             1 :1
FREEREC            0 :1
FREEBLK            0 :10
RSVBLK          NULL :
RSVIX              0 :
FPOOL           NULL :
DELTHR          NULL :
DYN             NULL :
```

Following the response to the DYN parameter, the following list is displayed. The list shows the details of how the indexed file will be constructed using the parameters just entered.

```
TOTAL LOGICAL RECORDS/DATA BLOCK:          3
FULL RECORDS/DATA BLOCK:                   2
INITIAL ALLOCATED DATA BLOCKS:            50
INDEX ENTRY SIZE:                         32
TOTAL ENTRIES/INDEX BLOCK:                 7
FREE ENTRIES/PIXB:                         1
RESERVE ENTRIES/PIXB(BLOCKS):              0
FULL ENTRIES/PIXB:                         6
RESERVE ENTRIES/SIXB:                      0
FULL ENTRIES/SIXB                          7
DELETE THRESHOLD ENTRIES:                  7
FREE POOL SIZE IN BLOCKS:                  0
# OF INDEX BLOCKS AT LEVEL   1:            9
# OF INDEX BLOCKS AT LEVEL   2:            2
# OF INDEX BLOCKS AT LEVEL   3:            1

DATA SET SIZE IN EDX RECORDS:             73
```

If a secondary file is being defined, the list of prompts is the same except for the following:

the reply to the prompt "SECONDARY INDEX (Y/N)?:" is Y

the secondary data set name is requested

the RECSIZE prompt is omitted; the Indexed Access Method computes the correct record size

```
SECONDARY INDEX (Y/N): Y
ENTER SECONDARY DATASET (NAME,VOLUME):
```

Parameter Descriptions for Option 2

The attributes of the file are determined by these SE command parameters:

BASEREC The estimated number of records to be initially loaded into the file in ascending key sequence. These records can be loaded by $IAMUT1 or by a PUT request after either a LOAD or PROCESS request.

If DYN is not specified, BASEREC defaults to null, resulting in an error condition. In this case, specify BASEREC as a positive number.

If DYN is specified, BASEREC defaults to one.

BLKSIZE The length, in bytes, of blocks in the file. It must be a multiple of 256. The Indexed Access Method uses 16 bytes in each block for a header.

RECSIZE The length, in bytes, of each record in the file. Record length must not exceed block length minus 16.

KEYSIZE The length of the key to be used for this file. The minimum key length is 1. For primary files, the maximum key length is 254. For secondary a secondary index, the maximum key length is 250.

KEYPOS The position, in bytes, of the key within the record. The first byte of the record is position 1.

FREEREC The number of free records to be reserved in each block. It must be less than the number of records per block (block size minus 16, divided by record size). If not, an error message is issued. The calculation is adjusted to ensure that there is at least one allocated record in the block; that is, there cannot be 100% free records.

FREEREC defaults to zero.

FREEBLK The percentage (0-99) of each cluster to reserve for free blocks. The percentage calculation result is rounded up so that at least one free block results. The calculation is adjusted to ensure that there is at least one allocated block in the cluster; that is, there cannot be 100% free blocks.

FREEBLK defaults to zero.

RSVBLK The percentage of the entries in each primary index block to reserve for cluster expansion. These reserved entries are used to point to new data blocks as they are taken from the free pool to expand the cluster. The result of the calculation is rounded up so that any non-zero specification indicates at least one reserved index entry. The calculation is adjusted to ensure that there is at least one allocated block in the cluster.

Enter a null character (&) for this prompt if you do not want initial reserved blocks and do not want the indexed access method to create reserved blocks as records are deleted and blocks become empty. Specify a value of zero for this prompt if you do not want initial reserved blocks but you do want the indexed access method to create reserved blocks as records are deleted and blocks become empty (See the DELTHR prompt).

Note that the sum of the FREEBLK and RSVBLK prompts must be less than 100 or an error message is issued. This value defaults to null if the DYN parameter is not specified. If the DYN parameter is specified, this value defaults to zero.

RSVIX The percentage (0-99) of the entries in each second level index block to reserve for use in case of cluster splits. A cluster split is required when there is no room to insert a new record in a cluster. Each cluster split uses one reserved entry of the second-level index block to create a new cluster with blocks from the free pool. The result of this calculation is rounded up so that any non-zero specification indicates at least one reserved index entry. The calculation is adjusted so that there is at least one unreserved entry in each second level index block. This value defaults to zero.

FPOOL The percentage (0-100) of the maximum possible free pool to allocate as determined by the RSVIX and RSVBLK parameters. The RSVBLK and RSVIX prompts result in a file structure set up to draw on the free pool for expansion.

If insertion activity is evenly distributed throughout the file, every reserve entry of every index block can be used. The number of blocks drawn from the free pool to support this unlikely condition is the maximum free pool size needed for the file. In more realistic cases, insertion activity is not evenly distributed throughout the file, so fewer free blocks are needed. The percentage specified here represents the evenness of the distribution of inserted records. Specify a large number (90, for example) if you expect insertions to be evenly distributed. Specify a small number (20, for example) if insertions are anticipated to be concentrated in specific key ranges.

If a null character (&) is specified for this prompt, a free pool is not created for this indexed file (you can use the DYN parameter to override this and create a free pool). If zero is specified, an empty free pool is created. Blocks can then be added to the free pool as records are deleted and blocks become empty (see the DELTHR prompt explanation). If you do not specify a null for this prompt, the RSVBLK must not be null and/or the RSVIX must be non-zero or an error is returned. Conversely, if the RSVBLK and/or RSVIX is non-zero, FPOOL must not be null or an error is returned.

The default for FPOOL is a null; no free pool is created.

DELTHR The percentage (0-99) of blocks to retain in the cluster as records are deleted and blocks made available. This is known as the delete threshold. When a block becomes empty, it is first determined if the block should be given up to the free pool by checking the response to this prompt. If the block is not given up to the free pool, it is retained in the cluster, either as a free block or as an active empty block. The result of this calculation is rounded up so that any non-zero specification indicates at least one block. The calculation is adjusted to ensure that the cluster always contains at least one block.

If the DELTHR parameter is specified as null (&) and DYN is not specified, DELTHR defaults to the number of allocated blocks in the cluster plus one half of the value calculated by the FREEBLK prompt. If the DELTHR parameter is specified as null and a value is specified for the DYN parameter, DELTHR defaults to zero.

DYN The number of blocks to be assigned to, or added to, the free pool. When DYN is used with other free pool parameters, the free pool size is calculated as specified by the FPOOL parameter plus the value specified for DYN.

If DYN is specified without the FPOOL parameter, the free pool is the number of blocks specified for DYN.

If DYN is specified, other parameters assume the following default values when specified as null:

```
BASEREC = 1
BLKSIZE = 0
RECSIZE = 0
KEYSIZE = 0
KEYPOS  = 1
FREEREC = 0
FREEBLK = 0
RSVBLK  = NULL
RSVIX   = 0
FPOOL   = NULL
DELTHR  = NULL
```

When you specify the number of blocks for the DYN parameter, remember that the Indexed Access Method can store several data records in a block, depending on the record size and block size you specify. Each block contains a 16 byte header. The number of records that can be contained in each block can be calculated by the following formula:

```
Records per block = (BLKSIZE-16)/RECSIZE
```

In the above calculation, use the integer quotient only; discard any remainder.

Blocks can be taken from the free pool for use as index blocks as well as for data blocks, so provide some extra blocks for these. A reasonable estimate of the number of index blocks required is 10%. Thus, if you know the number of data records you would like to add to the file, you can calculate the number of blocks to specify for the DYN parameter as follows:

$$\text{DYN} = \frac{\text{(Number of records to insert)} \times 1.1}{\text{(Records per block)}}$$

Option 3

Option 3 issues a prompt to determine what existing file to obtain the parameters from. The parameters can be set exactly according to the parameters of the original file by replying Y to the appropriate prompt. Otherwise, the parameters will be set based on the current condition of the existing data set to reflect insert activity.

```
SECONDARY INDEX (Y/N)?: N
NAME OF EXISTING INDEXED DATA SET (NAME,VOLUME): IAMFILE,EDX003
NEW PARAMETERS EXACTLY SAME AS ORIGINAL PARAMETERS (Y/N) ? Y

DATA SETSIZE IN EDX RECORDS:              17
INDEXED ACCESS METHOD RETURN CODE:        -1
SYSTEM RETURN CODE:                       -1
```

All Options

For all three options, the prompts are followed by the option of entering the DF (define file) function directly from the SE command. This simplifies the file definition process. The prompt is as follows:

```
 CREATE/DEFINE FILE (Y/N) ? Y
 ENTER DATASET (NAME,VOLUME): FILE01,EDX003
 NEW DATASET IS ALLOCATED
```

The immediate write-back option is then queried:

```
 DO YOU WANT IMMEDIATE WRITE-BACK? Y
```

The next prompt allows you the option of invoking the load or reorganize functions as follows:

```
INVOKE LOAD(L), REORGANIZE(R) OR END(E) AFTER CURRENT FUNCTION? L
DEFINE IN PROGRESS
```

Size calculations are performed using the parameter values you specify.

After the values are entered, the following is displayed showing the size and structure of the defined indexed file.

```
DATA SET SIZE IN EDX RECORDS:              17
INDEXED ACCESS METHOD RETURN CODE:         -1
SYSTEM RETURN CODE:                        -1

CREATE/DEFINE FILE (Y/N)?:
```

UN - UNLOAD INDEXED FILE

UN unloads an indexed file to a sequential file. Records are read from the indexed file with the Indexed Access Method GETSEQ request and written into the sequential file with the EDL WRITE instruction. If a secondary indexed file is specified, the primary file will be unloaded in secondary key sequence.

You can unload a secondary index independent of its primary if you first use the UE subcommand of the DR command of $IAMUT1 to set the independent indicator. You must turn the independent indicator off when the unload operation is completed.

The record lengths of the two files need not be the same. Unloaded records are truncated or padded with zeroes if the records lengths of the two data sets differ. For further detail, see the LO command.

Records are placed in the sequential file in ascending key sequence as indicated by the indexed file. Unloaded records can be blocked or unblocked. For a description of blocked and unblocked data sets, see "Blocked and Unblocked Sequential Data Sets" on page IG-164.

The UN command prompts you for the block size of the file to be unloaded. A null response or a value less than or equal to the record size causes the indexed file to be unloaded to an unblocked sequential file. The sequential file block size is calculated as the record size rounded up to the next 256-byte multiple value. If you want the file to be unloaded to a blocked sequential file, specify the actual block size value to the prompt "OUTPUT BLOCK SIZE". The record and block sizes of subsequent output sequential files are assumed to be the same as the initial output sequential file.

If the indexed file contains more records than are allocated in the sequential file, you are given the option to continue unloading to another sequential file. If you choose to continue unloading, you are prompted for the name of the file and volume to use to continue the unload operation. The unload operation continues, putting the records read from the indexed file into the new sequential file. If the end of the output file is reached and you choose not to continue, the unload operation ends.

Note: Do not specify the same file for input and output.

The following example shows the use of the UN command to put 80-byte records into a blocked sequential file.

```
ENTER COMMAND (?): UN

UNLOAD ACTIVE
ENTER INPUT DATASET (NAME,VOLUME): EDXF02,AM4VOL
ENTER OUTPUT DATASET (NAME,VOLUME): SEQ01,EDX003
OUTPUT RECORD ASSUMED TO BE 80 BYTES. OK?: Y
ENTER OUTPUT BLOCK SIZE (NULL = UNBLOCKED): 256
UNLOAD IN PROCESS

END OF INPUT DATASET
   100 RECORDS UNLOADED
UNLOAD SUCCESSFUL
ENTER COMMAND (?): EN
```

$IAMUT1 COMPLETION CODES

Completion Code	Condition
-1	Successful completion
7	Link module in use
8	Load error for $IAM
12	Data set shut down
13	Module not included in load module
23	Get storage error - IACB
31	FCB WRITE error during IDEF processing, check system return code
32	Blocksize not multiple of 256
34	Data set is too small
36	Invalid block size during file definition processing
37	Invalid record size
38	Invalid index size
39	Record size greater than block size
40	Invalid number of free records
41	Invalid number of clusters
42	Invalid key size
43	Invalid reserve index value
44	Invalid reserve block value
45	Invalid free pool value
46	Invalid delete threshold value
47	Invalid free block value
48	Invalid number of base records
49	Invalid key position
50	Data set is already opened for exclusive use
51	Data set opened in load mode
52	Data set is opened, cannot be opened exclusively
54	Invalid block size during PROCESS or LOAD
55	Get storage for FCB error
56	FCB READ error
60	LOAD mode key is equal to or less than previous high key in data set
61	End of file
62	Duplicate key found

Note: For completion codes number 37 through 49, check your parameters for consistency.

Completion Code	Condition
65	Inconsistent free space parameters, RSVIX/RSVBLK or FPOOL/DYN were specified without one or the other
100	READ error, check system return code
101	WRITE error, check system return code
110	WRITE error - data set closed
201	Request failed because the primary file for this secondary could not be opened. Check system return code
210	Request failed because $DISKUT3 could not be loaded
230	Directory read error from $IAMDIR, verify that directory exists
231	$IAMQCB not found, check sysgen for include of IIAMQCB
232	Directory open error for $IAMDIR, verify that directory exists
233	Directory related primary request is a primary entry
234	Directory error - DSNAME,VOL not found in $IAMDIR
235	Directory resource has not been requested
239	Directory write error. Refer to previously displayed message

CHAPTER 9. THE $VERIFY UTILITY

$VERIFY checks the validity of an indexed file and prints control block and free space information about the file on $SYSPRTR.

This $VERIFY description contains the following topics:

- $VERIFY Functions
- Invoking $VERIFY
- $VERIFY Example
- $VERIFY Messages
- $VERIFY Storage Requirements

$VERIFY FUNCTIONS

With $VERIFY you can:

- Verify that all pointers in an indexed file are valid and that the records are in ascending sequence by key.
- Verify the contents of a secondary index against the primary file and report any discrepancies.
- Print a formatted File Control Block (FCB) listing, including the FCB Extension block. The FCB Extension block contains the original file definition parameters.

  Note: The FCB Extension block does not exist and file definition parameters are not saved in the FCB for indexed files defined prior to version 1.2 of the Indexed Access Method. The reorganize (RO) $IAMUT1 command can be used to reformat those files by adding an FCB Extension block to make use of all the $VERIFY facilities.
- Print a report showing the distribution of free space in your file.
- Determine if any space is available for inserts.

INVOKING $VERIFY

$VERIFY can be invoked from either a terminal or a program coded in Event Driven Language. You supply the same input in either case. If you invoke $VERIFY from a terminal, supply the input required in response to prompts. If you invoke $VERIFY from a program, supply the input required as parameters passed to the program.

$VERIFY INPUT

This section describes the input required to execute $VERIFY.

name,volume

Data set and volume names for the primary index file or secondary index to be processed. (Endures that all chains within this data set are correct).

Option

The type of processing you want $VERIFY to do. The three options are:

Y - The FCB and the FCB Extension blocks are formatted and printed. The file is verified. A free space report is printed.

N - The FCB and the FCB Extension blocks are formatted and printed. The file is verified. No free space report is printed.

F - The FCB and the FCB Extension blocks are formatted and printed. No free space report is printed, but the '# OF AVAILABLE BLOCKS IN FREEPOOL' entry can be examined to determine if space is available for inserts; if the value is greater than zero (>0), space is available.

Cross verify option

The type of check you want $VERIFY to do between the primary index files and secondary indexes. The options are:

Y - a. If a primary index file was specified above as the data set name, this will check that all entries in the primary index file are in the secondary index.

b. If a secondary index was specified above as the data set name, this will check that all entries in the secondary index are in the associated primary indexed file.

N - Do not perform any cross verification.

secname,volume	Data set and volume names of the secondary index to be verified. Specify 'ALL' to verify all secondary indexes associated with the primary file.

INVOKING $VERIFY FROM A TERMINAL

Load the $VERIFY program as follows:

```
> $L $VERIFY
```

When $VERIFY begins execution, you are prompted for the parameters described previously. A complete example of a $VERIFY invocation from a terminal is shown under "$VERIFY Example" on page IG-192.

INVOKING $VERIFY FROM A PROGRAM

$VERIFY can be invoked by EDL programs with the LOAD instruction. The only required parameter is the address of a 30-byte area that contains:

	Hex Displacement	Length (Bytes)
Data set name	0	8
Volume name	8	6
Detail listing request (Y, N, or F)	E	1
Secondary file cross verify (Y or N)	F	1
Secondary index file name	10	8
Secondary index file volume	18	6

The next example shows the use of $VERIFY to verify a file named IAMFILE in the volume EDX002. A file verification and free space report are requested. The secondary file named SECIAM in the volume EDX002 is also verified.

```
EXAMPLE  PROGRAM START
START    EQU   *
          .
          .
          .
         LOAD  $VERIFY,PARMLIST,EVENT=VERIFY
         WAIT  VERIFY              WAIT FOR POST COMPLETE
          .
          .
          .
         PROGSTOP
PARMLIST EQU   *
DSNAME   DC    CL8'IAMFILE'        INDEXED DATA SET NAME
VOLUME   DC    CL6'EDX002'         VOLUME NAME
DETAIL   DC    CL1'Y'              PROCESSING OPTION
SECONDRY DC    CL1'Y'              SECONDARY FILE VERIFICATION
SECDSN   DC    CL8'SECIAM'         SECONDARY FILE NAME
SECVOL   DC    CL6'EDX002'         SECONDARY FILE VOLUME
VERIFY   ECB   -1                  EVENT CONTROL BLOCK
         ENDPROG
         END
```

$VERIFY EXAMPLE

This section presents the input and output for an example run of $VERIFY, along with descriptions of the material presented.

$VERIFY is invoked from the terminal as follows:

```
[1] > $L $VERIFY
[2] INDEXED ACCESS METHOD FILE VERIFICATION PROGRAM ACTIVE

[3] ENTER NAME OF OUTPUT PRINTER. (BLANK = $SYSPRTR):

[4] (NAME,VOLUME): DPRIM1,EDXIAM

[5] DO YOU WANT DETAIL LISTING? (Y/N/F/?)?: Y

[6] DO YOU WISH TO VERIFY SECONDARY VS PRIMARY INDEXES (Y/N):N

[7] VERIFICATION COMPLETE,     0   ERROR(S) ENCOUNTERED

[8] $VERIFY   ENDED
```

[1] In this example, the first line loads and executes $VERIFY.

[2] The second line is printed by the program to indicate that execution has begun.

[3] This line allows you to direct the output to a particular printer or terminal. You can also press the Enter key without supplying a device name and the output will be printed on $SYSPRTR.

[4] In the fourth line, the program prompts for the data set name and volume of the indexed file to be referenced by the program. In this example the reply indicates that the data set is DPRIM1, located on volume EDXIAM.

[5] In the fifth line, the program prompts for the amount of detail to be provided as output. The response of Y indicates that maximum detail is to be provided.

[6] In the sixth line, the program prompts for verification of secondary indexes. The response of N indicates that secondary indexes are not to be verified. As the program executes, it provides output to the printer, as shown in the example outputs that follow.

[7] Finally, messages are displayed to indicate the number of errors found. [8] This information message is provided stating that the program has ended.

FCB REPORT

The first page of the example output from $VERIFY follows. This page is always printed.

```
VERIFY REPORT. FILE = DPRIM1  , VOLUME = EDXIAM

FLAG1 :   FILE    FILE
         LOADED   TYPE
           Y       1   (0=PRPQ, 1=PP)
*****************************************
KEY SIZE =                                 6
KEY POSITION =                             1
BLOCK SIZE =                             256
RECORD SIZE =                             60
INDEX ENTRY SIZE =                        10
RBN OF HIGH LEVEL INDEX BLOCK IN USE =     2
RBN OF LAST DATA BLOCK IN USE =          786
RBN OF FIRST DATA BLOCK IN USE =           6
TOTAL RECORDS PER DATA BLOCK =             4
TOTAL ENTRIES PER INDEX BLOCK =           24
LOAD POINT VALUE FOR A DATA BLOCK =        4
LOAD POINT VALUE FOR AN INDEX BLOCK =     24
*****************************************
FLAG2 : IMMEDIATE   SECONDARY   FILE
       WRITE-BACK  INDEX FILE  FORMATTED
          N           EN          Y
*****************************************
VERSION NUMBER =                         2.0
DELETE THRESHHOLD (RECORDS) =              0
# OF AVAILABLE BLOCKS IN FREEPOOL         30
RBN OF 1ST FREE POOL BLOCK =             787
RBN OF HIGHEST LOGICAL INDEX BLOCK =       2
LEVEL OF HIGHEST INDEX BLOCK IN USE=       3
CURRENT NO. OF RECORDS IN FILE =        3000
```

The preceding page of the example report is interpreted as follows:

The first line shows the data set name and volume.

FLAG1: These three lines show the significant bits of the first flag byte in the FCB. The first two of the three lines are a heading. The third line shows the bit value (1 = on and 0 = off or Y = on and N = off). The headings are defined as follows:

FILE LOADED: Data set has been loaded flag. This flag is set when any record has been successfully loaded into the file in load mode.

FILE TYPE: This flag indicates whether the indexed file was created with the Realtime Programming System Indexed Access Method PRPQ (bit=0) or either the Event Driven Executive or Realtime Programming System Indexed Access Method Program Product (bit=1).

KEY SIZE: Shows the size of the key in bytes.

KEY POSITION: Shows the byte displacement of the key from the start of the record.

BLOCK SIZE: Shows the byte length of blocks in the file.

RECORD SIZE: Shows the byte length of records in the file.

INDEX ENTRY SIZE: Shows the number of bytes in each index entry. This length should be the key length plus 4, rounded up to a multiple of two bytes.

RBN OF HIGH LEVEL INDEX BLOCKK IN USE: Shows which index block is to be used as the starting point when the index is to be searched.

RBN OF LAST DATA BLOCK IN USE: Points to the last logical data block in the file which has been used.

RBN OF FIRST DATA BLOCK IN USE: Points to the first logical data block in the file which has been used. It is used as the starting point when a sequential read operation is begun with no key specified.

TOTAL RECORDS PER DATA BLOCK: Shows how many data records can be contained in a data block.

TOTAL ENTRIES PER INDEX BLOCK: Shows how many index entries can be contained in an index block.

LOAD POINT VALUE FOR A DATA BLOCK: The number of records that can be placed in each data block while in load mode. This value is calculated at file definition time to provide the requested number of free records.

LOAD POINT VALUE FOR AN INDEX BLOCK: The number of data blocks in each cluster to be used while in load mode. This value is calculated at file definition time to provide the space requested by the RSVBLK, RSVIX and FREEBLK parameters.

FLAG2: Another byte of flags described by a pair of lines: a heading line followed by a data line. The heading has the following meaning:

IMMEDIATE WRITE-BACK:	Immediate write back flag. If set (Y), this flag indicates that the immediate write back option was specified when the indexed file was defined.
SECONDARY INDEX FILE:	A Y indicates that this is a secondary file. N indicates that this is a primary file.
FILE FORMATTED:	Y indicates that the file has been formatted. N indicates that only the parameters have been specified and the file allocated. The file has not been formatted.

VERSION NUMBER: Shows the version number and modification level of the Indexed Access Method that was used to define the indexed file.

DELETE THRESHHOLD (RECORDS): Indicates the number of data blocks to retain in each cluster as records are deleted and blocks become empty. This value is calculated when the file is defined and is based on the DELTHR parameter.

# OF AVAILABLE BLOCKS IN FREEPOOL: The number of available blocks in the free pool. This count is updated as blocks are taken from or returned to the free pool.

RBN OF 1ST FREE POOL BLOCK: Points to the last block which was put in the free pool (which is the next block to be taken from the free pool).

RBN OF HIGHEST LOGICAL INDEX BLOCK: Points to the logical top of the index. In some cases (if the file has not been completely loaded), this RBN might not agree with the RBN OF HIGHEST LEVEL INDEX BLOCK IN USE. If it does not agree, then the file is structured with index blocks that are not yet needed because the file does not contain enough records.

LEVEL OF HIGHEST INDEX BLOCK IN USE: Indicates how many levels of the index are currently in use.

CURRENT NO. OF RECORDS IN FILE: The current number of records that are now contained in the file.

FCB EXTENSION REPORT

The second page of the example output from $VERIFY follows. This page is always printed.

This information is obtained from the FCB Extension block and shows the parameters that were specified when the file was defined. Some information (BLKSIZE, RECSIZE, KEYSIZE, KEYPOS) is duplicated on the FCB and FCB Extension report because it is contained in both control blocks. The values should correspond with each other. The word NULL for the value of a parameter indicates that no value was specified when the file was defined.

```
VERIFY REPORT. FILE = DPRIM1    ,   VOLUME = EDXIAM

INDEX FILE DEFINED WITH THESE PARAMETERS:

BASEREC=          3000
BLKSIZE=           256
RECSIZE=            60
KEYSIZE=             6
KEYPOS=              1
FREEREC=             0
FREEBLK=             0
RSVBLK=           NULL
RSVIX=               0
FPOOL=            NULL
DELTHR=           NULL
DYN=                30
```

Note: The parameters are the file definition parameters that were specified using the SE command of the $IAMUT1 utility when the file was defined.

FREE SPACE REPORT

A free space report of the example output from $VERIFY follows. The free space report is printed only if the $VERIFY option is specified as Y.

```
VERIFY REPORT. FILE = XMPL1   ,   VOLUME = EDX002
          TOTAL    USED    UNUSED  RESERVE  FREE    AVAILABLE    HIGH KEY
 RBN LVL ENTRIES ENTRIES ENTRIES ENTRIES BLOCKS RECORD SLOTS (FIRST
                                                             20 CHAR.
 --- --- ------- ------- ------- ------- ------ ------------ --------
   2   3      24       2       0      22      0          --   143949

   3   2      24      24       0      16      0          --   130536
   4   2      24       8       0      16      0          --   143949

   5   1      24      24       0       0      0           0   044932
  30   1      24      24       0       0      0           0   046750
  55   1      24      24       0       0      0           0   048655
  80   1      24      24       0       0      0           0   050527
 105   1      24      24       0       0      0           0   052392
 130   1      24      24       0       0      0           0   054225
 155   1      24      24       0       0      0           0   056075
 180   1      24      24       0       0      0           0   057930
 205   1      24      24       0       0      0           0   059829
 230   1      24      24       0       0      0           0   061640
 255   1      24      24       0       0      0           0   063548
 280   1      24      24       0       0      0           0   065389
 305   1      24      24       0       0      0           0   067297
 330   1      24      24       0       0      0           0   069166
 355   1      24      24       0       0      0           0   071029
 380   1      24      24       0       0      0           0   072887
 405   1      24      24       0       0      0           0   074731
 430   1      24      24       0       0      0           0   076586
 455   1      24      24       0       0      0           0   078441
 480   1      24      24       0       0      0           0   080329
 505   1      24      24       0       0      0           0   082175
 530   1      24      24       0       0      0           0   084006
 555   1      24      24       0       0      0           0   085861
 580   1      24      24       0       0      0           0   130536
 605   1      24      24       0       0      0           0   132395
 630   1      24      24       0       0      0           0   134205
 655   1      24      24       0       0      0           0   136097
 680   1      24      24       0       0      0           0   137929
 705   1      24      24       0       0      0           0   139815
 730   1      24      24       0       0      0           0   141655
 755   1      24      24       0       0      0           0   143523
 780   1      24       6       0      18      0           0   143949

VERIFICATION COMPLETE,     0   ERROR(S) ENCOUNTERED
```

In this report, each printed line represents an index block. The columns have the following meanings:

RBN: The relative block number within the indexed file, based on the block size specified when the file was defined. The first block in the file is relative block number zero.

LVL: The level of the index block analyzed. Lowest level (PIXB) is 1, second level (SIXB) is 2, etc.

TOTAL ENTRIES: The maximum number of index entries that can fit in an index block.

USED ENTRIES: The number of entries used in this index block.

UNUSED ENTRIES: The number of entries in the index block which are neither used nor reserved.

RESERVE ENTRIES: The number of reserve entries in this index block. This number represents the number of new index blocks that can be obtained from the free pool for creation of new blocks, provided there are enough blocks remaining in the free pool.

FREE BLOCKS: The number of free blocks associated with this index block.

AVAILABLE RECORD SLOTS: The maximum number of records that can be inserted into this cluster without obtaining blocks from the free pool.

HIGHEST KEY IN BLOCK: The first 20 bytes of the highest key in the block.

$VERIFY MESSAGES

As $VERIFY executes, any errors encountered result in an error message being written describing the type of error and where the error occurred.

FILE ERROR MESSAGES

The following messages indicate that the indexed file contains errors:

```
# BLOCKS IN FREEPOOL CHAIN DOES NOT MATCH FREE POOL COUNT IN FCB.

BLOCK OUT OF SEQUENCE. RBN _____.

HIGH KEY IN RBN _____ DOES NOT MATCH INDEX ENTRY IN RBN _____.

POINTERS IN HEADER OF HIGH INDEX BLOCK ARE NOT ZERO.

RBN _____ CONTAINS INVALID UPWARD POINTER.

RBN _____ CONTAINS INVALID BACKWARD POINTER.

RBN _____ CONTAINS INVALID FORWARD POINTER.

RBN _____ IS IN FREEPOOL CHAIN, BUT IS NOT A VALID FREEPOOL BLOCK.

RECORD OUT OF SEQUENCE NEAR RBN _____.

RECORD MATCH NOT FOUND FOR SEC INDEX.
PRIMARY=
SECONDARY=
```

If any of these messages are printed, the indexed file has at least one error.

Possible sources of the error include:

- The data set is not an indexed file
- Data in the file has been inadvertently destroyed
- Secondary index is not auto-update
- The Indexed Access Method has a program error

ERROR RECOVERY PROCEDURE

If any of the $VERIFY file error messages are printed, use the following procedure:

- Dump the file or portion of the file which $VERIFY indicated has errors.
- Attempt to reorganize the file with the $IAMUT1 utility RO command.

- If reorganization fails, submit an APAR, including the file dump.
- Secondary indexes may need to be regenerated. Invoke $VERIFY for each of the secondaries to determine if they are error free. If errors are indicated rebuild the index from the primary data sets after the problem has been corrected.

OTHER MESSAGES

Any other messages that are generated by $VERIFY can be found in the Messages and Codes.

$VERIFY STORAGE REQUIREMENTS

Working storage space is needed required for $VERIFY and the amount required varies, depending on the maximum number of blocks at the SIXB level and the block size of the file.

USING DEFAULT WORKING STORAGE REQUIREMENTS

The default working storage specification is 4K bytes. For a file with a block size of 256,this default is sufficient to handle up to 896 blocks at the SIXB level. The larger the block size of the file, the fewer the maximum number of SIXBs that can be processed.

The following formula can be used to calculate the maximum number of blocks at the SIXB level that $VERIFY can process, given the block size of the indexed file:

```
NS = (4096 - (2 * BLKSIZE)) / 4
```

NS is the number of blocks at the SIXB level
BLKSIZE is the block size of the indexed file

MODIFYING WORKING STORAGE REQUIREMENTS

The default working storage allocation is intended to satisfy the requirements of most indexed files. It may be necessary or desirable to modify the amount of working storage space available to $VERIFY.

The following formula can be used to calculate the amount of working storage required to process a file with a given block size and number of blocks at the SIXB level.

```
DS = (4 * NS) + (2 * BLKSIZE)
```

Where:
- DS is the amount of dynamic storage required
- NS is the number of blocks at the SIXB level
- BLKSIZE is block size of the indexed file

The number of SIXBs in a file can be determined by examining the free space report.

You can override the default working storage size at load time (if loaded by a program), or with the SS command of the $DISKUT2 utility.

SUMMARY

$VERIFY requires a variable amount of working storage which defaults to 4K bytes. Increase the working storage size if $VERIFY runs out of space during execution.

Decrease the working storage size if the number of SIXBs is significantly less than that supported by the default working storage allocation (896 with a block size of 256) and your available storage is limited.

CHAPTER 10. STORAGE AND PERFORMANCE CONSIDERATIONS

This chapter describes the storage required for the Indexed Access Method and suggestions for improving performance. The main topics are:

- Determining Storage Requirements
- Data Paging
- Other Performance Considerations

DETERMINING STORAGE REQUIREMENTS

The minimum amount of storage required by the Indexed Access Method is dependent upon the package you choose to install, plus the link module and any error exit routine you may have written. The approximate sizes of the available packages are included here for planning purposes.

THE INDEXED ACCESS METHOD PACKAGES

The Indexed Access Method program product is shipped with four packages:

- $IAM
- $IAMRS
- $IAMNP
- $IAMRSNP

You select the particular package to install on your system which meets your requirements for function, storage, and performance. The individual packages are described below:

1. $IAM - (18K), a full function Indexed Access Method package using overlay structure. It is expected to satisfy the needs of most users.

2. $IAMRS - (27K), a full function Indexed Access Method as a fully resident package. It requires more storage than $IAM, but offers maximum performance.

3. $IAMNP - (15K), this package is similar to $IAM (using an overlay structure) but does not include data paging. It is designed for users who have severe storage limitations.

4. $IAMRSNP - (24K), this package is similar to $IAMRS (fully resident) but does not include data paging. This package provides the perform-

ance of a resident system but is intended for users who do not have sufficient storage to take advantage of the data paging feature.

Notes:

1. The storage values above do not include Indexed Access Method control blocks, the central buffer (minimum of 2 X block size), and secondary index update buffers (minimum of 2 X record size).

2. To find the exact size of your Indexed Access Method package, load $IAM with the operator command $L. A message will be displayed about the loaded program. The number, followed by the letter P, indicates the size of the program in 256-byte pages. Multiplying this number by 256 yields the size in bytes of $IAM, including control blocks, work areas and buffers.

INDEXED ACCESS METHOD STORAGE ENVIRONMENT

A single copy of the Indexed Access Method load module $IAM serves the entire system.

Figure 12 on page IG-205 shows the components of the Indexed Access Method, and their relationship to the operating system.

The Indexed Access Method control blocks, buffers and programs are contained in a single module, which can be loaded in any partition (but only one copy on the system).

Application programs in any partitions (including the partition containing the Indexed Access Method) can invoke Indexed Access Method services using the IBM supplied link module, which must be included in the application program.

If the data paging feature of the Indexed Access Method is active, it uses storage in the partition(s) you select for performance improvement. This storage is in the form of a load module, $IAMSTGM.

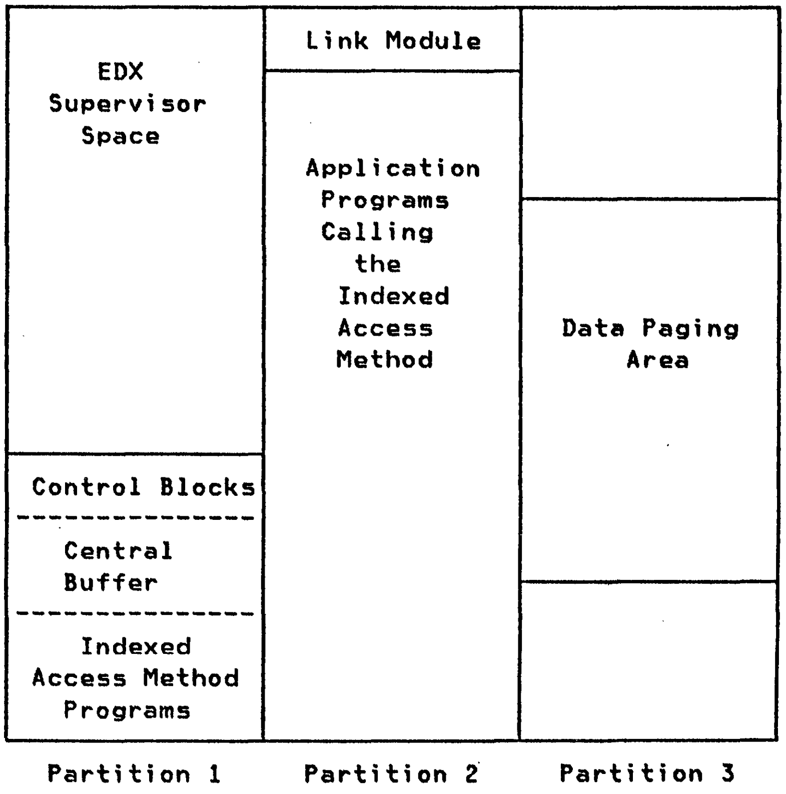


Figure 12. Indexed Access Method Storage Environment

Because $IAM is loaded automatically when the first Indexed Access Method request is issued, it does not need to be explicitly loaded before being used by any program. When loaded automatically on the first Indexed Access Method request, $IAM is loaded into partition 1 if enough storage is available there. If not, attempts are made to load $IAM into successively higher numbered partitions until space is found or no more partitions are available. Once loaded, the Indexed Access Method remains in storage until cancelled with the $C operator command.

The Indexed Access Method can also be loaded manually by using the $L operator command or automatically at IPL time through the automatic initialization capability (refer to the System Guide for a description of the automatic initialization capability). $IAM can be loaded into any partition, including partition one. It can be invoked (through the link module) from application program in any partition.

PERFORMANCE

Performance can be improved by various factors and the performance will be different for each application. One performance consideration has been described previously, the resident Indexed Access Method packages $IAMRS and $IAMRSNP. Another supplied performance feature is data paging.

DATA PAGING

Data paging is a performance feature that uses main storage space for a paging area (a cache) to improve the performance of the Indexed Access Method. This paging area retains recently used index and data blocks which have been retrieved for processing. As blocks are read from an indexed file, they are retained in the paging area on the assumption that they will probably be requested again. When a block is requested again, if it is in the paging area, no I/O operation is required; the block is moved directly into the central buffer.

The paging area is divided into 2K-byte (2048-byte) pages. Each indexed file can also be thought of as being divided into 2K-byte pages. When data is read from the file, a 2K-byte page is read and saved in the paging area. When data is written to the file, only the modified block (not the 2K-byte page) is written.

When the paging area becomes full, pages are overlaid according to a least-recently-used algorithm. The Indexed Access Method data paging algorithm handles direct access records differently from the way it handles sequential access records.

SEQUENTIAL ACCESS AND DATA PAGING: All of the pages in the page area can be used for direct access. However, because sequential access can cause the page area to be flushed out (negating the advantages of data paging), only 25% of the pages are set aside for use in sequential mode. Therefore, pages referenced in sequential mode will only use a small portion of the page area. This causes the pages to tend to preempt themselves instead of flushing out the page area.

REMOVAL OF STORAGE MODULES: The data paging area is obtained by loading a copy of $IAMSTGM into one or more partitions. Each copy of $IAMSTGM remains in storage, even if you cancel $IAM. Cancelling $IAM is not recommended unless you have ascertained that no files are currently open and no requests are about to be issued. If you have cancelled $IAM you can use the $C $IAMSTGM operator command to remove the data paging storage module from each partition. $IAMSTGM should never be cancelled until you have first cancelled $IAM.

Adjusting the Size of the Paging Area

Because every application is different you should not regard any information relative to the following described example as being directly applicable to your application. However, the general principles should apply to most applications.

Figure 13 on page IG-208 shows the effect of various data paging area sizes on the percentage of times a requested block was in the paging area ("Hit Ratio") and the resultant performance (response time indicator) for one application. The data was acquired by measuring the performance, and printing data paging statistic, while the application was running. The total size of all indexed files being accessed during the run was 36592 sectors (9.3M-bytes). It must be stressed that this is only one application, and your application may not behave in the same manner.

The three variables considered in data paging described in this example are:

- storage size dedicated to data paging
- the percentage of times that the block requested is in the paging area ("Hit Ratio")
- read/write ratio

STORAGE SIZE: The figure shows general trends for various storage sizes. Note that there is a minimum amount of storage which can provide a benefit. In this example the minimum storage to acquire a performance improvement is approximately 20k-bytes. This is because the data paging algorithms in the Indexed Access Method require a certain amount of processing, which is additional overhead. Your application may have a different minimum. If you cannot supply enough storage to provide a benefit, you are better off not to use data paging. Within certain limits, the more storage you supply, the better the performance. However, there are optimal minimum and maximum limitations. Figure 13 on page IG-208 shows that, for this example application, the minimum amount is about 20k-bytes.

The optimal maximum amount of storage, beyond which the benefit of using more storage becomes less pronounced, is about 70k-bytes for the example shown in Figure 13 on page IG-208. You must determine, based on your own storage/performance tradeoff requirements, how much storage to dedicate to data paging for the performance improvement you receive. Larger files require a proportionately larger paging area to attain the same hit ratio.

THE "HIT RATIO": The values shown at the left side of Figure 13 on page IG-208 is called a "Hit Ratio". This ratio is a percentage of how often an index block or data block requested is already in the paging area. Most applications tend to concentrate activity in a few areas of the file for a time, then move on to other areas of the file. These applications can use data paging to good advantage because there is a probability that the data being requested has been recently requested.

If your application references data in a completely random manner, data paging will be less efficient. Random applications result in a smaller hit ratio for a given paging area size than applications that concentrate on certain areas of the file. Therefore, larger paging area is required to obtain the same hit percentage.

THE READ/WRITE RATIO: The data paging function is optimized for read operations. In order to insure file integrity, write operations cause a write-through to the file. This means that there is no benefit in using data paging for write operations. In fact, due to paging overhead, write operations are less efficient with data paging than without data paging.

The higher your ratio of reads to writes, the more efficiently the data paging algorithm works, thus the better your performance improvement. In the example shown in Figure 13 on page IG-208, 80% of the requests were reads, 20% were writes.

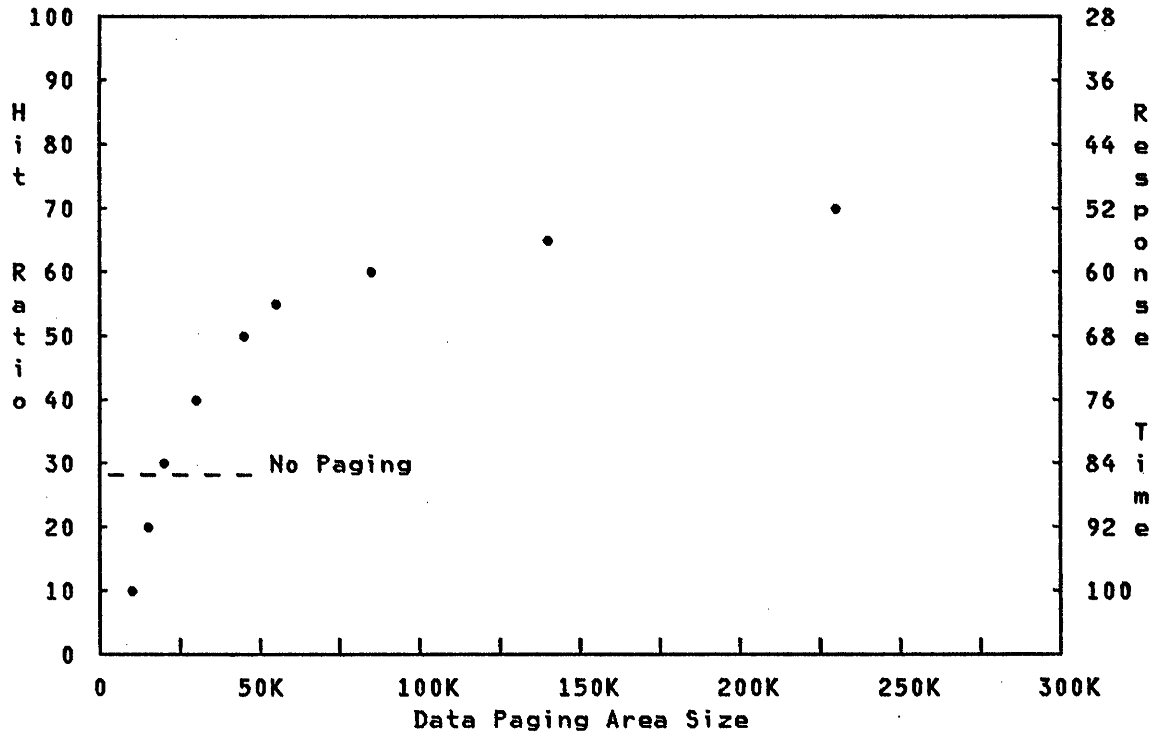


Figure 13. Plot of Data Paging Area Sizes. This graph shows how the size of the data paging area (shown across the bottom) affects the hit ratio (shown on the left margin), are the results in the response time (shown on the right margin). The unit of time for the response time scale is not given because it is application dependent. For this application, a hit ratio of at least 28% (which can be achieved with a paging area size of about 20K) is required to attain performance equal to that without data paging active. This is due to data paging processing overhead. Also note that a paging area size of greater than about 70K provides relatively little response improvement for the amount of storage dedicated.

Using Data Paging

The Indexed Access Method is distributed with the paging area size set to zero; therefore, the data paging function is not enabled. To use paging, use the $IAMUT1 PP command to set the paging area size for each partition and the $IAMUT1 PG command to activate paging.

When $IAM is loaded, the loader attempts to obtain storage in the requested partition. When storage is requested in a particular partition to activate paging, you are informed of the results with appropriate messages. The messages returned to inform you of the paging status are written to the $SYSLOG device. If $SYSLOG is not available, the messages are written to $SYSPRTR device. Following are the conditions which can result:

1. Data paging is successfully initialized. The storage you requested or the default amount of storage required for data paging is available. The following message is displayed:
   "DATA PAGING ACTIVE."

2. Data paging is unsuccesful.

   a. If you have attempted to activate paging and you requested zero for the storage amount, or the minimum amount of storage necessary for paging is not available in the partition you specified, the following message is displayed:
      "NOT ENOUGH STORAGE AVAILABLE FOR DATA PAGING."
      "DATA PAGING NOT ACTIVE."

   b. If you have requested more storage for paging than is available in the partition you specified, the following message is displayed
      LOAD FAILED FOR $IAMSTGM RC=xxx, PTN= Y, SIZE= zz

where:	xxx	represents the return code from the LOAD instruction
	y	represents the partition number requested
	zz	represents the size in 2048-bytes of storage you requested

OTHER PERFORMANCE CONSIDERATIONS

Following is a list of subjects followed by some ideas you might use to affect the performance of your application:

- Looking at the File Structure
- Controling the File size
- Reducing the Number of Index Levels
- Increasing the Buffer Size
- Avoiding Resource Contention

LOOKING AT THE FILE STRUCTURE: Performance of the Indexed Access Method is primarily determined by the structure of the indexed file being used. This structure is determined by parameters you specify when you create the file. The best performance from an indexed file is attained when the file structure is well planned and the free pool is rarely used, if it exists at all. For descriptions of the file parameters, see "Chapter 12. Installing the Indexed Access Method." For examples of the affects of parameter values, see "Chapter 3. Defining Primary Index Files."

Use the $IAMUT1 utility to see the effects of the various parameters on the file structure.

FILE SIZE: A large file spans more cylinders of the direct access device, so the average seek to get the record you want is longer. Splitting files into smaller files according to application type, or moving seldom used records to a "history file" might be viable solutions for file size reductions.

If your records contain unused or unnecessary fields, delete those fields and reduce your record length before defining and loading your file. The Sort/Merge Program Product contains facilities to accomplish this while sorting your records by key.

REDUCING THE NUMBER OF INDEX LEVELS: A file with many index levels requires more accesses to get to the desired data record, thus degrading performance. Factors which influence the number of index levels are:

- Number of records in file - see "File Size" previously described

- Amount and type of free space - see "File Structure" previously described

- Block size - when defining your indexed file, remember larger block sizes usually require fewer I/O operations

- Key size - shorter keys are more efficient than long keys. If only a portion of your key field provides uniqueness, set your key position and key length to that portion of the field when you define the file.

INCREASING THE BUFFER SIZE: The buffers required for I/O operations for all Indexed Access Method requests throughout the system are taken from a single buffer pool. The size can be changed at any time (to become effective during the next load of $IAM) as described under "RE - Reset Parameters" on page IG-170. If you provide a large buffer when you install the Indexed Access Method, it is more likely that blocks (especially high-level index blocks) needed are already in storage and need not be recalled from the file.

AVOIDING RESOURCE CONTENTION: Application programs that use the Indexed Access Method are executed the same as other application programs. Because the Indexed Access Method and the indexed files are resources available to all tasks, delays can occur. When more than one task uses the Indexed Access Method, contention can occur between tasks for any of the following resources:

- An entire indexed file
- An index block in the file
- A data block in the file
- A data record in the file
- Buffer space from the system buffer pool

For example, during the execution of a request from task A, some buffer space is required and an index block, data block, or record is locked (made unavailable to other requests). A request from task B requires more buffer space than is available or attempts to retrieve a block or record that was locked by task A. Task B must wait until the required resource becomes available.

Resources required by the Indexed Access Method are allocated only for the duration of a request except under the following circumstances:

- During an update, when control returns to the task after a GET or GETSEQ for update, the subject record is locked. The lock is released when the update is completed with a PUTUP, PUTDE, RELEASE, or DISCONN.

- During sequential processing, when control returns to the task after a GETSEQ, the block containing the subject record is locked and held in the buffer.

  Subsequent GETSEQ requests pick up records directly from the buffer. When a GET requires a record from the next block, the current block and buffer are released. Pending requests for a buffer area are satisfied and the next block is locked and held in the buffer. Except for momentary release of the buffer area between blocks, a block is locked while it is being processed. Processing is terminated by an end-of-data condition, an ENDSEQ request, a DISCONN request, or an error condition.

Use the following guidelines to avoid resource contention:

- Disconnect all indexed files before task termination. The DISCONN request releases locked records or blocks and writes records that have not already been written.

- Use conditional requests whenever possible so that your application can be productive while a resource is unavailable.

- Try to schedule applications so that they do not execute at the same time.

- If a file is used for "read only" by more than one application, consider multiple copies of the file using unique file names.

- With multiple Indexed Access Method applications, use direct access to retrieve a group of records. A suggested method is the following:

  1. Retrieve the first record by key.

  2. Extract the key from the record and save it for the next retrieval.

  3. Retrieve the next record using the saved key and a greater than key relational operator (GT or UPGT).

  4. Repeat the second and third steps until processing is complete.

Secondary Index Functions

Using secondary indexes affects the performance of the Indexed Access Method. Some of those reasons are described here.

DIRECT RETRIEVAL: Direct retrievals are somewhat slower when using a secondary index because of the extra accesses required to retrieve the data record from the primary file.

SEQUENTIAL RETRIEVAL: Sequential Retrievals are slower when using a secondary index because the records are returned in order by secondary key. The primary file containing the data records is in order by the primary key. Therefore, the records are not stored in the same sequence that they are retrieved. This requires random accesses to obtain the records.

RECORD INSERTS: Record inserts are slower if any associated secondary indexes have the auto-update indicator on. A new record must be inserted into each auto-update secondary index, as well as the primary, whether the original insert was a primary or a secondary.

RECORD DELETE: Record deletes are slower for the same reason as for inserts; records must be deleted from secondary indexes that have the auto-update indicator on. However, the impact for deletion is more severe than for insertion. This is because a search is required when multiple records have the same value for their secondary key as the record being deleted. The group of records having the same key must be sequentially searched until the record with the required primary key is found. This time could be quite significant if you have large groups of duplicate keys.

RECORD UPDATE: Record updates that modify the secondary key must also update any associated secondary index which has the auto-update indicator on. The secondary index is updated by deleting the old key and inserting the new key.

DATA RECORD MOVEMENT: Each record in a secondary index contains a pointer to the RBN where the record is located in the primary index file. If a data record has been moved, due to insert/delete activity in nearby areas of the primary file, the RBN in the secondary index record will be wrong. When the affected data record is retrieved through the secondary index, the error is detected. A full retrieval is then performed, using the primary key to obtain the data record. The RBN in the secondary index record is then updated for the benefit of future retrievals. This activity will affect the performance.

$VERIFY PERFORMANCE: The $VERIFY performance will be slower when the primary file being verified has a secondary index with large numbers of duplicate secondary keys. This is because the entire group of duplicate keys must be searched for the proper record. Because $VERIFY retrieves all records in the file, these impacts accumulate and the total execution time can be longer than expected.

CHAPTER 11. ERROR RECOVERY

This chapter describes how to handle Indexed Access Method errors and how to diagnose application program errors.

HANDLING ERRORS

All Indexed Access Method requests return a code in the task code word of the Task Control Block (TCB). The task code word is the same name as the task name. The return code reflects the condition of the requested function. Return codes are grouped in the following categories:

- -1 - Successful completion
- Positive - Error
- Negative (other than -1) - Warning

ERROR EXIT FACILITIES

There are three types of error exits for your application:

- Task error exit, provided by the supervisor
- Error exit, provided by the Indexed Access Method
- The task error exit used by the Indexed Access Method itself in case of an error

Task Error Exit

You can specify a task error exit routine that will receive control if your application program causes a soft exception or if a machine check occurs during the execution of your application.

Because your application may have requests pending (for example, a record is being held for update or a file is being processed sequentially), you should issue a DISCONN request before terminating your application. The task error exit allows you to release records, disconnect from any file you are connected to, and make your resources available to other applications. Use of the task error exit facility helps to ensure data integrity and allows proper termination or continuation of your application.

Implementing the task error exit facility is described in the System Guide.

Error Exit

In PROCESS and LOAD requests, the address of an error exit routine can be specified by the ERREXIT parameter. If specified, this routine is executed whenever an Indexed Access Method request terminates with a positive return code.

If the exit routine is not specified, the next sequential instruction after the request is executed regardless of the value of the return code.

$IAM Task Error Exit

The Indexed Access Method itself has a task error exit. If this error exit is given control by the supervisor, it writes these messages to the $SYSLOG device:

```
$IAM HAS INCURRED A SEVERE ERROR
$IAM CENTRAL BUFFER ADDRESS IS xxxx - PARTITION n
 PSW  LSB
 yyyy zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz zzzz
```

Where xxxx is the address, and n is the partition containing $IAM.

The PSW (yyyy), and LSB contents (zzzz), are also listed. For an explanation of the PSW (program status word) and the LSB (level status block), refer to the Operator's Reference.

$IAM then goes into an unrecoverable wait and will not process any access requests. You can dump the central buffer with the $D system command and take appropriate action to quiesce your application (refer to the Operator's Reference for a description of the $D command).

You can use the recovery and backup procedures, described under "File Backup and Recovery" on page IG-94, to restore the file, or you can resume execution of your application. To restart your application, you can either IPL or cancel $IAM and reload it.

If you wish to extend the logic of the error exit, code your own exit to replace the $IAM task error exit. Then rename CDIERR (the $IAM task error exit), name your error exit CDIERR, and rebuild $IAM.

SYSTEM FUNCTION RETURN CODES

If a system function called by an Indexed Access Method request terminates with a positive return code, the return code is placed in a location named by the SYSRTCD parameter in the PROCESS or LOAD request. This location is used until a DISCONN is issued.

For example, the GET request uses the supervisor read function. If the read terminates with a positive return code, that return code is saved in the location named by the SYSRTCD parameter in the PROCESS request associated with the GET request. The GET request also terminates with a positive return code in the task control word. The positive return code indicates that a read error has occurred. The cause of the read error can be determined by examining the location named by the SYSRTCD parameter.

> **Note:** When analyzing errors, the Indexed Access Method return code in the task code word would be checked prior to the system return code.

The following example is a method of obtaining the return code value from the location SYSRTCD. This routine gets the task SYSRTCD, and compares it to the EDX successful return code, negative one (-1).

```
YOURPRGM    PROGRAM  START
              .
              .
              .
            SUBROUT  ERRTEST
            MOVE     TASKRC,SYSRTCD     get system return code
            IF       (TASKRC,EQ,-1)     if -1, return now
              .
              .                         if not -1 then perform
              .                         your diagnosis
            ENDIF
            RETURN
              .
              .
   TASKRC   DATA     F'0'               saved system return cods
```

THE DATA-SET-SHUT-DOWN CONDITION

Sometimes an I/O error occurs that is not associated with a specific request. For example, task A issues a GET on file X. To secure buffer space to satisfy the request, the Indexed Access Method attempts to write a block to file Y and, in writing the record, an error occurs. Data set Y is damaged but there is no requesting program to accept an error return code.

The error is indicated by setting the data-set-shut-down condition for file Y. After this condition occurs, no requests except a DISCONN are accepted for file Y.

Later, if task B issues a GET on file Y, the request is terminated with a data-set-shut-down return code. Task B should issue a DISCONN and use recovery and backup procedures as described under "File Backup and Recovery" on page IG-94, to reconstruct the file. To cancel the data-set-shut-down condition, initial program load (IPL) or cancel $IAM.

$ILOG - ERROR LOGGING FACILITY

You can have the Indexed Access Method to log errors in the system error log data set.

To use the error logging ficility, allocate a system error log data set. The data set must be a minimum of 3 256-byte EDX records. The first two records are used for control, and this would allow one error log entry. The error log entries are entered in the log data set one after the other as they occur. When the data set becomes full, the new entries overlay the old entries starting at the front of the data set again. Therefore the size of the data set should be based on the frequency of errors, and the frequency with which the data set is listed or examined. Each error log entry requires a 256-byte EDX record.

Yoy can load $LOG into any partition. Using the system command LOGINIT, will initialize and activate error logging for any Indexed Access Method errors.

The following format is used to record Indexed Access Method associated errors in the error log data set:

Displacement into each EDX $IAM error log entry

Hex	(Dec)	Field
00	(0)	Indexed Access Method entry identifier 'CDIIAM'
06	(6)	User TCB Address
08	(8)	Data Set Name
10	(16)	Volume
16	(22)	Original Function Byte
17	(23)	Current Function Byte
18	(24)	Indexed Access Method Return Code
1A	(26)	System Return Code
1C	(28)	User AKR Value

To list the Indexed Access Method error log entries currently in the system error log, you can load $ILOG using the system command $L. Respond to the prompt "(DSNAME,VOLUME):" with system error log data set and volume name.

The list will be directed to the terminal which was used to load $ILOG.

Following is a sample of the printed error report showing two error records:

```
INDEXED ACCESS METHOD LOG REPORT PROGRAM ACTIVE

    TCB                           ORIG   CURR   $IAM   SYSTEM
PTN ADDR  DSNAME   VOLUME  FNCTN  FNCTN  RTCODE RTCODE  DATE     TIME
___ ____  _______  ______  _____  _____  ______ ______  ________ ________
 2  1F64  IAMFILE  EDX002  PUTNW  PUTNW    62     -1    00/00/00 00:00:00
 2  1F64  IAMFILE  EDX002  PUTNW  PUTNW    62     -1    00/00/00 00:00:00

  2 INDEXED ACCESS METHOD LOG ENTRIES LOCATED

$ILOG ENDED
```

DEADLOCKS AND THE LONG-LOCK-TIME CONDITION

Because the Indexed Access Method uses record and block locks to preserve file integrity, deadlock and long-lock-time conditions may occur.

The deadlock condition occurs when two or more tasks interact in such a way that one or more resources becomes permanently locked, making further progress impossible. A deadlock can also occur when two requests from the same task require a lock on the same record or a lock on the same block in sequential mode.

A long-lock-time condition occurs when your program acquires a record for update and does not return the record to $IAM for a long time.

Application tasks should avoid using the Indexed Access Method in such a way that a record or block remains locked for a long period of time, because other tasks may attempt to use the same record or block. In a terminal-oriented system, make every effort to ensure that a record or block is not locked during operator "think" time. Specifically, you should attempt to follow these rules:

- Do not retrieve a record for update, display the record at the terminal, and wait for the operator to modify it.

- Do not retrieve a record in sequential mode, display the record at the terminal, and wait for an operator response.

In both of these cases, a record or block is locked during operator "think" time and could be locked indefinitely.

A deadlock cannot be broken except by freeing the locks (records) that are being waited on.

If your application uses more than one IACB, deadlocks are possible. For example, one task has read record A and attempts to read record B, while another task has read record B and attempts to read record A. If you are using more than one IACB per task, such as in Multiple Terminal Manager applications, use ENQ/DEQ and interprogram communications to avoid the deadlocks.

You can avoid the long-lock-time condition by using one of the following two methods:

1.

   a. Retrieve the desired record without specifying update.

   b. Perform processing in a work area.

   c. Retrieve the record, specifying update.

   d. Compare the record read in step 1 with the record read in step 3.

e. If the records are identical, issue a PUTUP request, specifying the address of the copy in the work area. If they are not identical, issue a RELEASE request for the record read in step 3, and repeat steps 1 through 5.

2. Use conditional requests which do not wait for locks. (See "Chapter 7. Coding the Indexed Access Method Requests" for descriptions of coding conditional requests.)

   To retrieve records in sequential mode, use the technique described in "Avoiding Resource Contention" on page IG-210.

VERIFYING REQUESTS AND FILES

Following are two steps you can take to help you isolate and correct malfunctions in your Indexed Access Method application program.

- Request verification - to determine that requests are correct check all parameters specified or defaulted on the Indexed Access Method CALL statements:

  - PROCESS/LOAD requests - When issuing a PROCESS or LOAD, check that the specified file name is the correct file control block (DSCB) for the file you are verifying.

  - GET-PUT-DELETE-RELEASE requests - For these requests, carefully check the key, its position, length, and the relational operator (if used). Ensure that the correct address for the indexed access control block (IACB) is passed from the PROCESS or LOAD request, and that the record area address is correct.

- File verification - read your $VERIFY report or indexed file dump to determine whether data or index records are missing or incorrect.

Note: Be sure that the combination of parameters specified by the SE command of the $IAMUT1 utility to define your file is correct (see the "Chapter 8. The $IAMUT1 Utility" on page IG-141 for a description of the $IAMUT1 parameters).

CHAPTER 12. INSTALLING THE INDEXED ACCESS METHOD

This chapter presents an overview of how to install the Indexed Access Method.

The Indexed Access Method is distributed on two double surface diskettes with external ID AM4001- and AM4002-V1M2BASVOL01. The diskettes are formatted are 256 bytes per sector.

INSTALLATION PROCEDURES

The installation information which follows is for planning purposes only. The specific details for installing the product is included in the "Program Directory", which is shipped with the product.

INSTALLING THE INDEXED ACCESS METHOD

Installing the Indexed Access Method consists of two steps:

1. Step 1
   a. Ensure that adequate space is available for the installation according to the requirements shown in Figure 14.

Volume	Data Sets	EDX Records	Contents
EDX002	3	265	Load Modules
ASMLIB	2	145	Source Modules
ASMLIB	1	6	Link Module

Figure 14. Volume Space Requirements

2. Step 2

   a. Copy the Indexed Access Method load module ($IAM), the utility program ($IAMUT1), the file verification program ($VERIFY), and the log report program ($ILOG), to the EDX002 volume.

   b. Copy the following source modules to the ASMLIB volume.

      - IAMEQU
      - FCBEQU
      - IAM

ASSEMBLING AND EXECUTING THE INSTALLATION VERIFICATION PROGRAM

To assemble and execute the installation verification program:

1. Submit to the $JOBUTIL utility, the 'proc' $SAMPROC provided on volume AM4001 to assemble and link edit the verification program.

   The source statements for the installation verification program are contained in a data set named SAMPLE on volume AM4001.

2. Use $IAMUT1 to define and allocate an indexed file to be used by the installation verification program. Respond to the SE option 2 prompts with the indicated values:

BASEREC	10	FREEBLK	10
BLKSIZE	256	RSVBLK	0
RECSIZE	80	RSVIX	0
KEYSIZE	28	FPOOL	0
KEYPOS	1	DELTHR	0
FREEREC	1	DYN	10

3. Load $SAMPLE and when prompted for the data set and volume, respond with the name for the file allocated in the previous step (SAMPFILE).

Note: The procedure $SAMPROC assumes that ASMWORK and LINKWORK data sets exist on EDX002. Allocate these data sets if they don't already exist with the $DISKUT1 Event Driven Executive Utility. (Refer to the Operator's Reference for details on allocation of these data sets.)

APPENDIX A. SUMMARY OF CALCULATIONS

The following calculations can be used to define an indexed data set. For a more detailed description of these calculations, see "Chapter 3. Defining Primary Index Files" on page IG-17. In the calculations requiring division, results with non-zero remainders are either:

truncated $\boxed{T}$, or rounded up $\boxed{U}$.

To truncate is to drop the remainder; to round up is to add one (only if the remainder is non-zero), and truncate.

Data Block		
[1] Records per data block = block size minus 16, divided by record size; result truncated.	[1] = (BLKSIZE-16)/RECSIZE	$\boxed{T}$
[2] Free records per block.	[2] = FREEREC	
[3] Allocated records per data block = Records per block minus free records per block.	[3] = [1] - [2]	
Index Block (General)		
[4] Index entry size = key length plus 4; must be even; add 1 if odd.	[4] = KEYSIZE + 4 (+1 if odd)	
[5] Total entries per index block = block size 16, divded index entry size; result truncated.	[5] = (BLKSIZE-16) / [4]	$\boxed{T}$
Index Block (PIXB)		
[6] Free entries per primary index block (PIXB) = specified percentage of total entries per index block; result rounded up.	[6] = FREEBLK % of [5]	$\boxed{U}$

[7] Reserve entries per PIXB = specified percentage of total entries per index block; result rounded up. If free entries per PIXB and reserve entries per all PIXB entries, subtract one from reserve entries per PIXB.	[7] = RSVBLK % of [5] (-1 if [6] + [7[= [5])	$\boxed{U}$
[8] Allocated entries per PIXB = total entries per index block minus free entries per PIXB, minus reserve entries per PIXB.	[8] = [5] - [6] - [7]	

<u>Index Block (SIXB)</u>

[9] Reserve entries per secondary index block (SIXB) = specified percentage of total entries per index block; relult rounded up. If reserve entries per SIXB require all SIXB entries, subtract one.	[9] = RSVIX % of [5] (-1 if [9] = [5])	$\boxed{U}$
[10] Allocated entries per SIXB = total entries per index block minus reserve entries per SIXB.	[10] = [5] - [9]	

<u>Delete Threshold</u>

[11] The number of blocks to retain in cluster (delete threshold) is calculated in one of three ways:

a. If the RSVBLK parameter was not specified: Number of blocks to retain in cluster = total entries per index block.	[11] = [5] or	
b. If the RSVBLK parameter was specified, but the DELTHR parameter was not specified: Number of blocks to retain in the cluster = allocated entries per PIXB, plus one-half of free entries per PIXB; rusult rounded up.	[11] = [8] + [6] / 2 or	$\boxed{U}$

c. If the RSVBLK parameter was specified, and the DELTHR parameter was specified: Number of blocks to retain in the cluster = specified percentage of total entries per index block; result rounded up. If the result is zero, set it to 1.

[11] = DELTHR % of [5] $\boxed{U}$

(If 0, set [11] to 1)

Data in Data Set

[12] Initial allocated data blocks = base records divided divided by allocated records records per data block; result rounded up.

[12] = BASEREC / [3] $\boxed{U}$

[13] Number of clusters in data set = initial allocated data blocks, divided by allocated entries per PIXB; result rounded up.

[13] = [12] / [8] $\boxed{U}$

[14] Total number of free blocks in data set = number of clusters in data set, times free entries per PIXB.

[14] = [13] * [6]

Indexes in Data Set

[15] Number of primary index blocks (PIXBs) = number of clusters in data set.

[15] = [13]

[16]aNumber of secondary index blocks (SIXBs) = number of PIXBs, divided by allocated entries per SIXB; result rounded up.

[16] = [15] / [10] $\boxed{U}$

[17]aCalculate the number of index blocks for levels 3 to n. Note that levels 1 (PIXB) and 2 (SIXB) have already been calculated. When the number of index blocks at a level is 1, n has been reached and the calculation is finished.

Number of index blocks at level i (i=3 to n) = number of index blocks at next lower level, divided by total entries per index block; result rounded up.	[17] = [17] / [5]	$\boxed{\text{U}}$
[18] Total number of index blocks = sum of index blocks at each level until a level containing asingle index block is attained.	[18] = [15] + [16] + (Sum of all [17]s)	
Free Pool		
[19] Number of new data blocks which can be assigned to existing clusters = reserve entries per PIXB, times number of PIXBs.	[19] = [7] * [15]	
[20] Number of new clusters (PIXBs) which can be crested = reserve entries per SIXB, times number of SIXBs.	[20] = [9] * [16]	
[21] Number of new data blocks which can be assigned to new clusters = total entries per index block, times number of new clusters which can be created.	[21] = [5] * [20]	
[22] Maximum possible free pool = number of new data blocks data blocks which can be assigned to existing clusters, plus number of new clusters (PIXBs) which can be created, plus number of new data blocks which can be assigned to new clusters.	[22] = [19] + [20] + [21]	
[23] Actual number of free pool blocks = specified percentage of maximum possible free pool; result rounded up.	[23] = FPOOL % of [22]	$\boxed{\text{U}}$

Size of Data Set

[24] Total number of blocks in data set = 1 (for file control block), plus total number of index blocks, plus initial allocated data blocks blocks, plus total number of free blocks in data set, plus actual number of free pool blocks.	[24] = 1 + [18] + [12] + [14] + [23]

APPENDIX B. PREPARING INDEXED ACCESS METHOD PROGRAMS

To prepare an application program that issues Indexed Access Method requests, perform the following steps:

1. Enter your source program statements, using one of the Event Driven Executive text editors ($FSEDIT, $EDIT1, or $EDIT1N).

2. Create the $EDXLINK control statements required to combine your program with $IAM (the link module) and any other object modules you may need in your application. Use one of the text editors to perform this operation.

3. Assemble or compile your source program.

4. Use the linkage editor, $EDXLINK, to combine the object modules into a single load module, using the control statements prepared in Step 2.

When the preceding steps are completed, the program is ready to be executed.

Refer to the Program Preparation Guide for more information on program preparation.

A SAMPLE $JOBUTIL PROCEDURE AND LINK-EDIT CONTROL DATA SET

The following are examples of a $JOBUTIL procedure and a link-edit control file used to prepare a program.

Sample $JOBUTIL Procedure

The following $JOBUTIL procedure is an example of preparing an EDL program.

```
*********************************************************
*
* THESE STATEMENTS WILL COMPILE, LINK, AND UPDATE THE
* APPLICATION.
*
*********************************************************
JOB      COMPILE
***   COMPILE USERPROG SOURCE  ***
LOG      $SYSPRTR
PROGRAM  $EDXASM,ASMLIB
DS       USERPROG,EDX002        SOURCE MODULE
DS       ASMWORK,EDX002         ASSEMBLER WORK DATA SET
DS       USEROBJ,EDX002         ASSEMBLER OUTPUT
PARM     LIST     $SYSPRTR
EXEC
JUMP     END,GT,4
JOB      LINK
LOG      $SYSPRTR
PROGRAM  $EDXLINK,EDX002
DS       LINKWORK,EDX002        WORK DSNAME
*                               LINK-CONTROL DATA SET
PARM     LINKCNTL,EDX002        &SYSPRTR
EXEC
LABEL    END
EOJ
```

Link Edit Control Data Set Example

The following link-edit control records can be used to link-edit an Indexed Access Method application with the Indexed Access Method.

```
*********************************************************
*
* LINK EDIT CONTROL DATA SET (LINKCTL)
*
*********************************************************
 INCLUDE USEROBJ,EDX002  INCLUDE APPLICATION PGM OBJECT
 INCLUDE IAM,ASMLIB      INCLUDE INDEXED ACCESS METHOD
 LINK    USERPROG,EDX002 REPLACE END
 END
```

APPENDIX C. CODING EXAMPLES

This chapter demonstrates how to code the Indexed Access Method request functions by means of sample programs. This example uses Event Driven Language CALL functions. The second example uses the COBOL language. The third example in this chapter is coded using PL/I language.

EDL INDEXED ACCESS METHOD CODING EXAMPLE

This program gives an example for each of the Indexed Access Method function calls. The indexed file is opened first in load mode and ten base records are loaded followed by a DISCONNECT. Next, the same file is opened for processing. A GET request is performed for the first record whose key is greater than 'JONES PW'. Two more records are retrieved sequentially and then the ENDSEQ call releases the file from sequential mode. A record is then retrieved directly by key and updated. Another record is retrieved sequentially and deleted. A new record is inserted and another one is deleted by their unique keys. Finally, an example of extracting information from the file control block is shown. Upon successful completion the message "Verification Complete" is displayed on the console.

Although using secondary keys are not demonstrated in this example the requests are coded the same for secondary keys as they are for primary keys. When accessing secondary keys use the secondary index file name instead of the primary index file name. The Indexed Access Method will open the primary index file and retrieve the data record according to the secondary key requested.

This program requires that an Indexed Access Method file has been defined with the $IAMUT1 utility with the following specifications:

BASEREC	10
BLKSIZE	256
RECSIZE	80
KEYSIZE	28
KEYPOS	1
FREEREC	1
FREEBLK	10
RSVBLK	0
RSVIX	0
FPOOL	0
DELTHR	0
DYN	0

EDL INDEXED ACCESS METHOD CODING EXAMPLE

```
SAMPLE   PROGRAM START,DS=??,ERRXIT=TEECB
START    EQU   *
*
         ENQT
         PRINTEXT LOGON,LINE=0  PRINT LOGON MESSAGE
         DEQT
*
*  OPEN THE INDEXED ACCESS METHOD DATA SET FOR LOADING
*
        CALL  IAM,(LOAD),IACB,(DS1),(OPENTAB),(SHARE)
*
*  LOAD THE INDEXED ACCESS METHOD DATA SET
*
        MOVEA POINTER,RECORD1          POINTER <== A(RECORD1)
        DO    RECNUM,TIMES
              CALL  IAM,(PUT),IACB,(*),P4=POINTER
              ADD POINTER,80           POINT TO NEXT RECORD
        ENDDO
*    GET OUT OF LOAD MODE
        CALL  IAM,(DISCONN),IACB
        EJECT
*
*  OPEN THE INDEXED FILE FOR PROCESSING
*
        CALL  IAM,(PROCESS),IACB,(DS1),(OPENTAB),(SHARE)
*
* PERFORM A DIRECT RETRIEVAL OF THE FIRST RECORD WHOSE KEY IS
* GREATER THAN 'JONES PW'.  THE KEY FIELD WILL BE MODIFIED TO
* REFLECT THE KEY OF THE RECORD RETRIEVED.
*
        CALL  IAM,(GET),IACB,(BUFF),(KEY3),(GT)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
*
*  PERFORM A SEQUENTIAL RETRIEVAL OF THE FIRST TWO RECORDS
*  WHOSE KEYS ARE GREATER THAN OR EQUAL TO 'JONES PW'
*
        CALL  IAM,(GETSEQ),IACB,(BUFF),(KEY1),(GE)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
        CALL  IAM,(GETSEQ),IACB,(BUFF)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
        CALL  IAM,(ENDSEQ),IACB,(BUFF)    END SEQUENTIAL MODE
```

```
*  UPDATE THE RECORD WHOSE KEY IS 'JONES PW' BY A DIRECT UPDATE
*
         CALL   IAM,(GET),IACB,(BUFF),(KEY1),(UPEQ)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
*
*  MAKE THE DESIRED MODIFICATIONS TO THE RECORD NOW IN BUFFER
*
         MOVE   BUFF+30,0
         CALL   IAM,(PUTUP),IACB,(BUFF)
*
*  DELETE THE RECORD WHOSE KEY IS 'JONES PW' BY A
*  SEQUENTIAL UPDATE
*
         CALL   IAM,(GETSEQ),IACB,(BUFF),(KEY1),(UPEQ)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
         CALL   IAM,(PUTDE),IACB,(BUFF)
         CALL   IAM,(ENDSEQ),IACB            END SEQUENTIAL MODE
*
*  INSERT A NEW RECORD WITH A KEY OF 'MATHIS GR'
*
         CALL   IAM,(PUT),IACB,(NEWREC)
*
*  DELETE THE RECORD WHOSE KEY IS 'LANG LK'
*
         CALL   IAM,(DELETE),IACB,(KEY2)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
         EJECT
*
*  EXTRACT THE FILE CONTROL BLOCK INTO THE EXTRACT BUFFER
*
         CALL   IAM,(EXTRACT),IACB,(EXTBUF),(FCBSIZE),128
         MOVEA  #1,EXTBUF        #1  <--  A(EXTRACT BUFFER)
         MOVE   FLAGBYTE,(0,#1),BYTE   OBTAIN FCB FLAG BYTE
         SPACE 5
*
*  WRITE VERIFICATION COMPLETE MESSAGE TO THE OPERATOR
*
         ENQT
               PRINTEXT SKIP=1
               PRINTEXT VERIF,SPACES=0
         DEQT
         GOTO   FINISH       JUMP AROUND ERROR ROUTINES
SYSERR   EQU    *            GETS CONTROL ON SYSIPGM CHECK
```

```
* WHEN A TASK ERROR EXIT IS SPECIFIED IN AN INDEXED
* ACCESS METHOD PROGRAM, YOU CAN RELEASE ALL ACTIVE
* RECORD AND BLOCK LEVEL LOCKS AS WELL AS DISCONNECT
* THE FILE ITSELF BY ISSUING THE 'DISCONN' CALL FOR
* EACH FILE THAT IS OPEN.
*
        GOTO  FINISH
        EJECT
IAMERR  EQU   *                GETS CONTROL UPON INDEXED
*                              METHOD ERRORS
        MOVE  RTCODE,SAMPLE
        ENQT
              PRINTEXT SKIP=2
              PRINTEXT RTCODMSG
              PRINTNUM RTCODE,TYPE=S,FORMAT=(3,0,I)
              PRINTEXT SKIP=1
              PRINTEXT ERRMSG,SPACES=0
        DEQT
FINISH  EQU   *
        CALL  IAM,(DISCONN),IACB
        PROGSTOP
        EJECT
*
*  DATA DEFINITION AND STORAGE AREAS
*
RECNUM   DATA   F'10'          NUMBER OF RECORDS TO LOAD
RTCODE   DATA   F'0'           INDEXED ACCESS METHOD RETURN CODE
OPENTAB  DATA   F'0'           SYSTEM RETURN CODE ADDRESS
         DATA   A(IAMERR)      ERROR EXIT ROUTINE ADDRESS
         DATA   F'0'           END OF DATA ROUTINE ADDRESS
RECORD1  DATA   CL80'BAKER RG'
RECORD2  DATA   CL80'DAVIS EN'
RECORD3  DATA   CL80'HARRIS SL'
RECORD4  DATA   CL80'JONES PW'
RECORD5  DATA   CL80'JONES TR'
RECORD6  DATA   CL80'LANG LK'
RECORD7  DATA   CL80'PORTER JS'
RECORD8  DATA   CL80'SMITH AR'
RECORD9  DATA   CL80'SMITH GA'
RECORD10 DATA   CL80'THOMAS SN'
FLAGBYTE DATA   H'0'                      FCB FLAG BYTE
         DATA   H'0'
```

```
NEWREC   DATA   CL80'MATHIS GR'
BUFF     DATA   CL80' '
KEY1     TEXT   'JONES PW',LENGTH=28
KEY2     TEXT   'LANG LK',LENGTH=28
         DATA   X'1C'
         DATA   X'00'
KEY3     DATA   CL28'JONES PW'
IACB     DATA   F'0'                ADDR OF IACB PUT HERE
EXTBUF   DATA   64F'0'              FCB PUT HERE BY EXTRACT
LOGON    TEXT   'INSTALLATION VERFICATION PROGRAM ACTIVE'
VERIF    TEXT   'VERIFICATION COMPLETE'
ERRMSG   TEXT   'VERIFICATION INCOMPLETE DUE TO BAD RETURN CODES'
RTCODMSG TEXT   'INDEXED ACCESS METHOD RETURN CODE: '
         EJECT
*
*  THE FOLLOWING STORAGE IS USED BY TASK ERROR EXIT HANDLING
*
TEECB    EQU   *           TASK ERROR EXIT CONTROL BLOCK
         DATA  F'2'        # OF DATA WORDS THAT FOLLOW
         DATA  A(SYSERR)   ADDRESS OF EXIT ROUTINE
         DATA  A(HSA)      ADDRESS OF HARDWARE STATUS AREA
*
*  HARDWARE STATUS AREA.  THIS STORAGE WILL BE FILLED IN BY
*  HARDWARE UPON SYSTEM OR PROGRAM CHECK INTERRUPT
*
HSA      EQU   *
         DATA  F'0'        PROCESSOR STATUS WORD
HSALSB   EQU   *           LEVEL STATUS BLOCK:
         DATA  F'0'        ADDRESS KEY REGISTER
         DATA  F'0'        INSTRUCTION ADDR REGISTER
         DATA  F'0'        LEVEL STATUS REGISTER
         DATA  8F'0'       GENERAL REGISTERS 0-7
         COPY  IAMEQU
         COPY  FCBEQU
         ENDPROG
         END
```

COBOL INDEXED ACCESS METHOD CODING EXAMPLE

This coding example inserts, deletes, and updates records in an indexed file, using primary and secondary keys to retrieve the records. The indexed file description is described below under "Input File".

Program Description

This program reads a record and based on a transaction code, either updates, deletes, or inserts records to a current Indexed Access Method file. The transaction type also determines whether indexing is done using a secondary or primary key.

Input File

```
        I. TRANSACTION FILE.

             TRANSACTION RECORD FORMAT:
             EMPLOYEE NUMBER      1-6     (6)
             LAST NAME            7-21   (15)
             FIRST NAME           22-31  (10)
             ADDRESS              32-56  (25)
             CITY                 57-68  (12)
             STATE                69-70   (2)
             AGE                  71-72   (2)
             START DATE           73-78   (6)
             TYPE                 79      (1)
             ACTION               80      (1)

       II. UPDATE FILE

          A. MASTER FILE.
             PRIMARY. KEY IS EMPLOYEE NUMBER
          B. NAME FILE
             SECONDARY. KEY IS LAST NAME.
             EMPLOYEE RECORD FORMAT
             EMPLOYEE NUMBER      1-6     (6)
             LAST NAME            7-21   (15)
             FIRST NAME           22-31  (10)
             ADDRESS              32-56  (25)
             CITY                 57-68  (12)
             STATE                69-70   (2)
             AGE                  71-72   (2)
             START DATE           73-78   (6)
             FILLER               79-80   (2)
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                 COBOL1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.            IBM-S1.
OBJECT-COMPUTER.            IBM-S1.
SPECIAL-NAMES.
    SYSOUT IS PRINTER.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT EMPLOYEE-MASTER ASSIGN TO DS2 "EMPMAST" "EDXTST"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS RANDOM
        RECORD KEY IS EMPLOYEE-NUMBER
        FILE STATUS IS SK.
    SELECT EMP-NAME-FILE ASSIGN TO DS3 "EMPNAME" "EDXTST"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS RANDOM
        RECORD KEY IS E-LAST-NAME
        FILE STATUS IS SK.
    SELECT TRANSACTION-FILE ASSIGN TO DS4 "TRANSF" "EDXTST"
        ORGANIZATION IS SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL
        FILE STATUS IS SK.
DATA DIVISION.
FILE SECTION.
FD  EMPLOYEE-MASTER
    LABEL RECORDS ARE STANDARD
    BLOCK CONTAINS 3 RECORDS.
01  MASTER-RECORD.
    05  EMPLOYEE-NUMBER    PICTURE X(06).
    05  FILLER             PICTURE X(74).

FD  EMP-NAME-FILE
    LABEL RECORDS ARE STANDARD
    BLOCK CONTAINS 3 RECORDS.
01  EMP-NAME-RECORD.
    05  FILLER             PICTURE X(06).
    05  E-LAST-NAME        PICTURE X(15).
    05  FILLER             PICTURE X(59).


FD  TRANSACTION-FILE
    LABEL RECORDS ARE STANDARD
    BLOCK CONTAINS 3 RECORDS
    RECORD CONTAINS 80 CHARACTERS.
01  TRANS-ACTION-RECORD     PICTURE X(80).
```

```
WORKING-STORAGE SECTION.
77  EOF                     PICTURE 9(01) VALUE ZERO.
77  ERR-SWITCH              PICTURE X(01) VALUE ZERO.
77  SK                      PICTURE X(02) VALUE ZERO.
01  EMPLOYEE-RECORD.
    05  MAN-NUMBER          PICTURE X(06).
    05  NAME.
        10  LAST-NAME       PICTURE X(15).
        10  FIRST-NAME      PICTURE X(10).
    05  STREET-ADDRESS      PICTURE X(25).
    05  CITY                PICTURE X(12).
    05  STATE               PICTURE X(02).
    05  AGE                 PICTURE X(02).
    05  START-DATE          PICTURE X(06).
    05  FILLER              PICTURE X(02).

01  TRANSACTION-RECORD.
    05  T-NUMBER        PICTURE X(06).
    05  T-NAME.
        10  T-LAST-NAME     PICTURE X(15).
        10  T-FIRST-NAME    PICTURE X(10).
    05  T-STREET-ADDRESS    PICTURE X(25).
    05  T-CITY              PICTURE X(12).
    05  T-STATE             PICTURE X(02).
    05  T-AGE               PICTURE X(02).
    05  T-START-DATE        PICTURE X(06).
    05  TRANSACTION-CODE.
        10  TRANS-TYPE      PICTURE X(01).
        10  TRANS-ACTION    PICTURE X(01).

PROCEDURE DIVISION.
BEGIN-PROCESSING.
    PERFORM FILE-OPEN1.
    IF SK  = "00"
      PERFORM PROCESS-SECTION UNTIL EOF = 1.
    PERFORM CLOSE-UP.
    DISPLAY " CLOSE UP PROC COMPLETE"  UPON PRINTER.
    STOP RUN.

PROCESS-SECTION.
    READ TRANSACTION-FILE INTO TRANSACTION-RECORD
    AT END MOVE 1 TO EOF
    DISPLAY "TRANSACTION FILE PROCESSING COMPLETE" UPON PRINTER.
    IF EOF NOT = 1
        DISPLAY " " TRANSACTION-RECORD UPON PRINTER
        PERFORM CONTROL-SECTION.
```

```
CONTROL-SECTION.
    IF  TRANS-ACTION NOT = "2"
        PERFORM READ-SECTION.
    IF TRANS-ACTION = 1
        PERFORM DELETE-PROC
    ELSE
        IF TRANS-ACTION = 2
            PERFORM ADD-PROC
        ELSE
            PERFORM UPDATE-PROC.
READ-SECTION.
    IF  TRANS-ACTION  = "P"
      PERFORM READ-PRIMARY
    ELSE
      PERFORM READ-SECONDARY.

READ-PRIMARY.
    DISPLAY " DURING PRIMARY READ" UPON PRINTER.
    DISPLAY " KEY= " T-NUMBER UPON PRINTER.
    MOVE T-NUMBER TO EMPLOYEE-NUMBER.
    READ EMPLOYEE-MASTER INTO EMPLOYEE-RECORD
       INVALID KEY DISPLAY "INVALID PRIMARY KEY" T-NUMBER
       UPON PRINTER.
    IF SK NOT = "00"
      DISPLAY "PRIMARY READ FAILED " T-NUMBER " " SK UPON
      PRINTER.

READ-SECONDARY.
      DISPLAY " DURING SECONDARY READ" UPON PRINTER.
      DISPLAY " KEY= " T-LAST-NAME UPON PRINTER.
      MOVE T-LAST-NAME TO E-LAST-NAME.
      READ EMP-NAME-FILE INTO EMPLOYEE-RECORD
         INVALID KEY DISPLAY "INVALID SECONDARY KEY"
         T-LAST-NAME UPON PRINTER.
      IF SK NOT = "00"
        DISPLAY "SECONDARY READ FAILED" T-LAST-NAME " " SK
        UPON PRINTER.

UPDATE-PROC.
    IF T-LAST-NAME NOT = SPACES
        MOVE T-LAST-NAME TO LAST-NAME.
    IF T-FIRST-NAME NOT = SPACES
    IF T-STREET-ADDRESS NOT = SPACES
        MOVE T-STREET-ADDRESS TO STREET-ADDRESS.
    IF T-CITY NOT = SPACES
        MOVE T-CITY TO CITY.
    IF T-STATE NOT = SPACES
        MOVE T-STATE TO STATE.
    IF T-AGE NOT = SPACES
        MOVE T-AGE TO AGE.
    IF T-START-DATE NOT = SPACES
        MOVE T-START-DATE TO START-DATE.
    IF TRANS-TYPE = "P"
      PERFORM PRIMARY-REWRITE
    ELSE
      PERFORM SECONDARY-REWRITE.

PRIMARY-REWRITE.
        DISPLAY " BEGIN PRIMARY REWRITE  KEY = " T-NUMBER
                UPON PRINTER.
        MOVE T-NUMBER TO EMPLOYEE-NUMBER.
        REWRITE MASTER-RECORD FROM EMPLOYEE-RECORD
               INVALID KEY DISPLAY "INVALID PRIMARY KEY"
               T-NUMBER UPON PRINTER.
        IF SK NOT = "00"
            DISPLAY "PRIMARY WRITE FAILED" T-NUMBER " " SK
            UPON PRINTER.
        DISPLAY " PRIMARY REWRITE COMPLETE" UOPN PRINTER.

SECONDARY-REWRITE.
        DISPLAY " BEGIN SECONDARY REWRITE  KEY = " T-LAST-NAME
               UPON PRINTER.
        MOVE T-LAST-NAME TO E-LAST-NAME.
        REWRITE EMP-NAME-RECORD FROM EMPLOYEE-RECORD
               INVALID KEY DISPLAY "INVALID SECONDARY KEY"
               " " T-LAST-NAME UPON PRINTER.
        IF SK NOT = "00"
             DISPLAY "SECONDARY WRITE FAILED" T-LAST-NAME " "
             SK UPON PRINTER.
        DISPLAY " SECONDARY REWRITE COMPLETE" UPON PRINTER.

DELETE-PROC.
    IF TRANS-TYPE = "P"
      PERFORM PRIMARY-DELETE
    ELSE
      PERFORM SECONDARY-DELETE.
```

```
PRIMARY-DELETE.
        MOVE TRANS-ACTION-RECORD TO MASTER-RECORD.
        DISPLAY " PRIMARY DELETE STARTED" UPON PRINTER.
        DELETE EMPLOYEE-MASTER RECORD
               INVALID KEY DISPLAY "INVALID PRIMARY KEY"
               T-NUMBER UPON PRINTER.
        IF SK NOT = "00"
            DISPLAY "PRIME KEY FOR DELETE NOT FOUND"
            UPON PRINTER.
        DISPLAY " PRIMARY DELETE FINISHED" UPON PRINTER.

SECONDARY-DELETE.
        MOVE TRANS-ACTION-RECORD TO EMP-NAME-RECORD.
        DISPLAY " SECONDARY DELETE STARTED" UPON PRINTER.
        DELETE EMP-NAME-FILE RECORD
               INVALID KEY DISPLAY "INVALID SECONDARY KEY"
               T-LAST-NAME UPON PRINTER.
        IF SK NOT = "00" DISPLAY
            "SECONDARY KEY FOR DELETE NOT FOUND" UPON PRINTER
            DISPLAY TRANSACTION-RECORD UPON PRINTER.
        DISPLAY " SECONDARY DELETE FINISHED" UPON PRINTER.

ADD-PROC.
    MOVE TRANSACTION-RECORD TO EMPLOYEE-RECORD
    IF TRANS-TYPE = "P"
      PERFORM PRIMARY-ADD
    ELSE
      PERFORM SECONDARY-ADD.

PRIMARY-ADD.
        WRITE MASTER-RECORD FROM EMPLOYEE-RECORD
               INVALID KEY DISPLAY "INVALID PRIMARY KEY"
               T-NUMBER UPON PRINTER.
        IF SK NOT = "00" DISPLAY
            "INSERT FAILED FOR PRIME FILE" UPON PRINTER
            DISPLAY TRANSACTION-RECORD UPON PRINTER.

SECONDARY-ADD.
        WRITE EMP-NAME-RECORD FROM EMPLOYEE-RECORD
               INVALID KEY DISPLAY "INVALID SECONDARY KEY"
               T-LAST-NAME UPON PRINTER.
        IF SK NOT = "00"
            DISPLAY "INSERT FAILED FOR SECONDARY FILE"
            UPON PRINTER.
```

```
FILE-OPEN1.
    OPEN I-O EMPLOYEE-MASTER.
    IF SK NOT = "00"
        DISPLAY "OPEN FAILED FOR EMPMAST" SK UPON PRINTER
    ELSE
        OPEN I-O EMP-NAME-FILE
        IF SK NOT = "00"
            DISPLAY "OPEN FAILED FOR EMPNAME" SK UPON PRINTER
        ELSE
            OPEN INPUT TRANSACTION-FILE
            IF SK NOT = "00"
                DISPLAY "OPEN FAILED FOR TRANSACTION-FILE" SK
                        UPON PRINTER.
        DISPLAY " FILE OPEN COMPLETE" UPON PRINTER.

CLOSE-UP.
    DISPLAY " BEGIN CLOSE UP PROC " UPON PRINTER.
    CLOSE TRANSACTION-FILE.
    CLOSE EMP-NAME-FILE.
    IF SK NOT = "00"
        DISPLAY "CLOSE FAILED FOR EMPNAME, RC= " SK
        UPON PRINTER.
    ELSE
        DISPLAY " EMP-NAME-FILE CLOSED " UPON PRINTER.
    CLOSE EMPLOYEE-MASTER.
    IF SK NOT = "00"
        DISPLAY "CLOSE FAILED FOR EMPMAST, RC= " SK
        UPON PRINTER.
    ELSE
        DISPLAY " EMP-MAST-FILE CLOSED " UPON PRINTER.
```

PL/I INDEXED ACCESS METHOD CODING EXAMPLE

This PL/I coding example inserts, deletes, and updates records in an indexed file, using primary and secondary keys to retrieve the records. The indexed file description is described below under "Input File".

Program Description

This program reads a record and based on a transaction code, either updates, deletes, or inserts records to a current Indexed Access Method file. The transaction code also determines whether index access is done using a secondary key or primary key.

Input File.

```
      I. INPUT FILE
         A. TRANSACTION FILE,
            TRANSACTION RECORD FORMAT:

            EMPLOYEE NUMBER      1-6     (6)
            LAST NAME            7-21   (15)
            FIRST NAME           22-31  (10)
            ADDRESS              32-56  (25)
            CITY                 57-68  (12)
            STATE                69-70   (2)
            AGE                  71-72   (2)
            START DATE           73-78   (6)
            TYPE                 79      (1)
            ACTION               80      (1)


     II. UPDATE FILE
         A. MASTER FILE.
            PRIMARY. KEY IS EMPLOYEE NUMBER
         B. NAME FILE
            SECONDARY. KEY IS LAST NAME.
            EMPLOYEE RECORD FORMAT
            EMPLOYEE NUMBER      1-6     (6)
            LAST NAME            7-21   (15)
            FIRST NAME           22-31  (10)
            ADDRESS              32-56  (25)
            CITY                 57-68  (12)
            STATE                69-70   (2)
            AGE                  71-72   (2)
            START DATE           73-78   (6)
            FILLER               79-80   (2)
```

```
PL1TEST: PROCEDURE OPTIONS(MAIN);
 DCL EMPMAST                        /* EMPLOYEE MASTER FILE      */
             FILE RECORD            /* PRIMARY                   */
             DIRECT                 /* KEY IS EMPLOYEE NUMBER    */
             UPDATE
             KEYED
             ENV(FB BLKSIZE(256) RECSIZE(80) INDEXED
                 KEYLENGTH(6) KEYLOC(1));

 DCL EMPNAME                        /* EMPLOYEE NAME FILE        */
             FILE RECORD            /* SECONDARY                 */
             DIRECT                 /* KEY IS EMPLOYEE NAME      */
             UPDATE
             KEYED
             ENV(FB BLKSIZE(256) RECSIZE(80) INDEXED
                 KEYLENGTH(15) KEYLOC(7));

 DCL TRANSFL                        /* TRANSACTION FILE          */
             FILE RECORD            /* INPUT FILE                */
             SEQUENTIAL
             INPUT
             ENV(FB BLKSIZE(240) RECSIZE(80) CONSECUTIVE);

 DCL SYSPRINT                       /* STANDARD OUTPUT FILE      */
             FILE PRINT
             ENV(F BLKSIZE(121));

 DCL 1 $FCBLST                      /* FILE CONTROL BLOCK LIST   */
          STATIC
          EXTERNAL,
       2 $FCBCNT                    /* FILE COUNT                */
            FIXED BIN(15)
            INIT(4),
       2 $FCBF1                     /* FILE #1 NAME              */
            CHAR(8)
            INIT('EMPMAST'),
       2 $FCBD1                     /* FILE #1 DATA SET          */
            CHAR(8)
            INIT('EMPMAST'),
       2 $FCBV1                     /* FILE #1 VOLUME            */
            CHAR(6)
            INIT('EDXTST'),
       2 $FCBF2                     /* FILE #2 NAME              */
            CHAR(8)
            INIT('EMPNAME'),
       2 $FCBD2                     /* FILE #2 DATA SET          */
            CHAR(8)
            INIT('EMPNAME'),
      2 $FCBV2                        /* FILE #2 VOLUME          */
            CHAR(6)
            INIT('EDXTST'),
      2 $FCBF3                        /* FILE #3 NAME            */
            CHAR(8)
            INIT('TRANSFL'),
      2 $FCBD3                        /* FILE #3 DATA SET        */
            CHAR(8)
            INIT('TRANSFL'),
      2 $FCBV3                        /* FILE #3 VOLUME          */
            CHAR(6)
            INIT('EDXTST'),
      2 $FCBF4                        /* FILE #4 NAME            */
            CHAR(8)
            INIT('SYSPRINT'),
      2 $FCBD4                        /* FILE #4 DEVICE NAME     */
            CHAR(8)
            INIT('SYSPRINT'),
      2 $FCBTL4                       /* FILE #4 TOP LINE        */
            FIXED BIN(15)
            INIT(1),
      2 $FCBBL4                       /* FILE #4 BOTTOM LINE     */
            FIXED BIN(15)
            INIT(66),
      2 $FCBHL4                       /* NOT USED                */
            FIXED BIN(15)
            INIT(00);

DCL 1 EMP_RECORD                      /* IAM BUFFER FORMAT       */
      STATIC,
      2 EMP_NUMBER
           CHAR(6),
      2 NAME,
        3 LAST_NAME
              CHAR(15),
        3 FIRST_NAME
              CHAR(10),
      2 STREET_ADDRESS
           CHAR(25),
      2 CITY
           CHAR(12),
      2 STATE
           CHAR(2),
      2 AGE
           CHAR(2),
      2 START_DATE
           CHAR(6),
      2 FILLER
           CHAR(2);
```

```
DCL 1 TRANSACTIONR                /* TRANSACTION RECORD FORMAT */
      STATIC,
      2 TEMP_NUMBER
          CHAR(6),
      2 TNAME,
        3 TLAST_NAME
            CHAR(15),
        3 TFIRST_NAME
            CHAR(10),
      2 TSTREET_ADDRESS
          CHAR(25),
      2 TCITY
          CHAR(12),
      2 TSTATE
          CHAR(2),
      2 TAGE
          CHAR(2),
      2 TSTART_DATE
          CHAR(6),
      2 TRANSACTION_CODE,
        3 TRANS_TYPE          /* 'P' = PRIMARY  'S' = SECONDARY */
            CHAR(1),
        3 TRANS_ACTION        /* 1 = DELETE  2 = INSERT         */
            CHAR(1);          /* 3 = UPDATE                     */
DCL BLANK                     /* FIELD OF BLANKS                */
      STATIC
      CHAR (6)
      INIT('      ');

DCL (IOERR,                   /* INPUT/OUTPUT ERROR FLAG        */
     FOERR)                   /* OPEN ERROR FLAG                */
       STATIC
       CHAR(1)
       INIT('F');

DCL TRUE                      /* TRUE VALUE FOR FLAGS           */
      STATIC
      CHAR(1)
      INIT('T'),
    FALSE                          /* FALSE VALUE FOR FLAGS     */
      STATIC
      CHAR(1)
      INIT('F');

DCL R_CODE                         /* RETURN CODE               */
      STATIC
      FIXED BIN(15)
      INIT(0);
```

```
 DCL ONCODE                        /* ON CONDITION CODE         */
       BUILTIN;
 DCL EOF                           /* END OF FILE FLAG          */
       STATIC
       CHAR(1)
       INIT('F');
/************************** MAIN PROGRAM **********************/
/*                                                             */
/*********************** ON CONDITION FOR EOF ******************/
  ON ENDFILE(TRANSFL)
    BEGIN;
      PUT LIST('*** TRANSACTION FILE PROCESSING COMPLETE ***');
      EOF = TRUE;
      CLOSE                        /* CLOSE ALL FILES           */
       FILE(EMPMAST),
       FILE(EMPNAME),
       FILE(TRANSFL);
      STOP TASK;
    END;
/*                                                             */
/**************** ON CONDITIONS FOR FILE OPEN ERRORS ************/
  ON UNDF(EMPMAST) FOERR = TRUE;
  ON UNDF(EMPNAME) FOERR = TRUE;
  ON UNDF(TRANSFL) FOERR = TRUE;
/*                                                             */
/******************** ON CONDITIONS FOR I/O ERRORS **************/
  ON KEY(EMPMAST) IOERR = TRUE;
  ON KEY(EMPNAME) IOERR = TRUE;
/*                                                             */
/*************************** OPEN ALL FILES ********************/
/*                                                             */
  CALL OPEN;
/******* INITIATE PROCESSING UNTIL EOF CONDITION IS REACHED *****/
DO WHILE (EOF ¬= TRUE);
  IOERR = FALSE;
  FOERR = FALSE;
  CALL PROCESS;      /* INVOKE PROCESS SUBROUTINE */
END;   /* END DO WHILE */
/********************* END MAIN PROGRAM ************************/
/*                                                             */
OPEN: PROC;
  OPEN FILE(EMPMAST) UPDATE;
  IF FOERR = TRUE THEN
    DO;
      R_CODE = ONCODE;                     /* SET RETURN CODE */
      PUT LIST('OPEN FAILED FOR EMPMAST') SKIP;
      PUT LIST('ON CODE = ',R_CODE) SKIP;
    END;
```

```
   IF FOERR = FALSE THEN
     DO;
       OPEN FILE(EMPNAME) UPDATE;
         IF FOERR = TRUE THEN
           DO;
             R_CODE = ONCODE;                /* SET RETURN CODE */
             PUT LIST('OPEN FAILED FOR EMPNAME') SKIP;
             PUT LIST('ON CODE = ',R_CODE) SKIP;
           END;
     END;
    /*                                                     */
   IF FOERR = FALSE THEN
     DO;
       OPEN FILE(TRANSFL);
         IF FOERR = TRUE THEN
           DO;
             R_CODE = ONCODE;                /* SET RETURN CODE */
             PUT LIST('OPEN FAILED FOR TRANSFL') SKIP;
             PUT LIST('ON CODE = ',R_CODE);
           END;
     END;
    /*                                                     */
   IF FOERR = TRUE
     THEN
       STOP TASK;
    /*                                                     */
 END; /* END OPEN PROCEDURE */
 /********************** PROCESS PROCEDURE ***********************/
 /*                                                              */
 /*    1) READS IN A TRANSACTION RECORD                          */
 /*    2) IF ACTION = 1 DELETES RECORD WITH CORRESPONDING KEY.   */
 /*                 = 2 INSERTS RECORD ONTO IAM FILE.            */
 /*                 = 3 READS RECORD WITH CORRESPONDING KEY,     */
 /*                   ALLOWS UPDATE, REWRITES RECORD.            */
 /*    3) IF TYPE = 'P' ALL INDEXING IS DONE WITH A PRIMARY KEY  */
 /*               = 'S' ALL INDEXING IS DONE WITH A SECONDARY KEY*/
 /*    4) ALL IDENTIFIERS, FILES AND RECORDS USED ARE GLOBAL     */
 /*                                                              */
 /****************************************************************/
 PROCESS: PROCEDURE;
    READ FILE(TRANSFL) INTO (TRANSACTIONR);
    IF IOERR = TRUE THEN
      DO;
        R_CODE = ONCODE;
        PUT LIST('READ HAS FAILED FOR TRANSFL') SKIP;
        PUT LIST('ON CODE = ',R_CODE) SKIP;
      END;
   IF TRANS_ACTION = '1' & IOERR = FALSE THEN
     CALL DELETE;                        /* BEGIN DELETE        */
   IF TRANS_ACTION = '3' & IOERR = FALSE THEN
     CALL UPDATE;
   IF TRANS_ACTION = '2' & IOERR = FALSE THEN
     CALL INSERT;

END;  /* END PROCEDURE PROCESS */

DELETE: PROC;
   IF TRANS_TYPE = 'P'
     THEN
       DELETE FILE(EMPMAST) KEY(TEMP_NUMBER);
     ELSE
       DELETE FILE(EMPNAME) KEY(TLAST_NAME);
 END; /* END DELETE */
 /*                                                             */
 UPDATE: PROC;
    IF TRANS_TYPE = 'P'
      THEN
         CALL PRIM_READ;
      ELSE
         CALL SEC_READ;
      IF IOERR = FALSE THEN
        DO;
          IF TLAST_NAME ¬= BLANK
            THEN
              LAST_NAME = TLAST_NAME;
          IF TFIRST_NAME ¬= BLANK
            THEN
              FIRST_NAME = TFIRST_NAME;
          IF TSTREET_ADDRESS ¬= BLANK
            THEN
              STREET_ADDRESS = TSTREET_ADDRESS;
          IF TCITY ¬= BLANK
            THEN
              CITY = TCITY;
          IF TSTATE ¬= BLANK
            THEN
              STATE = TSTATE;
          IF TAGE ¬= BLANK
            THEN
              AGE = TAGE;
          IF TSTART_DATE ¬= BLANK
            THEN
              START_DATE = TSTART_DATE;
          CALL REWRITE;

   END;                             /* END UPDATE              */
END;
/*                                                             */
PRIM_READ: PROC;
   READ FILE(EMPMAST) INTO(EMP_RECORD)
        KEY(TEMP_NUMBER);
   IF IOERR = TRUE THEN
     DO;
       R_CODE = ONCODE;
       PUT LIST('EMPMAST PRIMARY READ HAS FAILED') SKIP;
       PUT LIST('KEY = ',TEMP_NUMBER);
       PUT LIST('ONCODE = ',R_CODE);
     END;
END;                         /* END PRIMARY READ               */
/*                                                             */
SEC_READ: PROC;
    READ FILE(EMPNAME) INTO(EMP_RECORD)
         KEY(TLAST_NAME);
    IF IOERR = TRUE THEN
      DO;
        R_CODE = ONCODE;
        PUT LIST('EMPMAST SECONDARY READ HAS FAILED') SKIP;
        PUT LIST('KEY = ',TLAST_NAME);
        PUT LIST('ONCODE = ',R_CODE);
      END;
  END;                        /* END SECONDARY READ             */
REWRITE: PROC;
   IF TRANS_TYPE = 'P'                  /* BEGIN REWRITE        */
     THEN
       REWRITE FILE(EMPMAST) FROM(EMP_RECORD)
               KEY(TEMP_NUMBER);
     ELSE
       REWRITE FILE(EMPNAME) FROM(EMP_RECORD)
               KEY(TLAST_NAME);
END;
```

```
 INSERT: PROC;
    IF TRANS_TYPE = 'P'
      THEN
        DO;
          WRITE FILE(EMPMAST) FROM(TRANSACTIONR)
                KEYFROM(TEMP_NUMBER);
          IF IOERR = TRUE THEN
            DO;
              R_CODE = ONCODE;
              PUT LIST('EMPMAST SECONDARY INSERTION HAS FAILED')
                SKIP;
              PUT LIST('KEY = ',TEMP_NUMBER);
              PUT LIST('ONCODE = ',R_CODE);
            END;
        END;
      ELSE
        DO;
          WRITE FILE(EMPNAME) FROM(TRANSACTIONR)
                KEYFROM(TLAST_NAME);
          IF IOERR = TRUE THEN
            DO;
              R_CODE = ONCODE;
              PUT LIST('EMPMAST SECONDARY INSERTION HAS FAILED')
                SKIP;
              PUT LIST('KEY = ',TNAME);
              PUT LIST('ONCODE = ',R_CODE);
            END;
        END;
  END;
END PL1TEST;
```

BIBLIOGRAPHY

This bibliography lists and summarizes the Event Driven Executive Version 3 library; it should assist you in using the library as a whole as well as direct you to the individual books you require.

See the publication IBM Series/1 Publications Directory, GC34-0361 for a complete list of Series/1 hardware and programming manuals. This publication lists each version of the Event Driven Executive and its related licensed programs, and shows which manuals apply to each version.

EVENT DRIVEN EXECUTIVE LIBRARY

The IBM Series/1 Event Driven Executive library consists of nine full-sized books, six specialized internals books, and a quick reference pocket book:

- IBM Series/1 Event Driven Executive System Guide, SC34-1702 (or System Guide)
- IBM Series/1 Event Driven Executive Operator's Reference, SC34-1703 (or Operator's Reference)
- IBM Series/1 Event Driven Executive Language Reference, SC34-1706 (or Language Reference)
- IBM Series/1 Event Driven Executive Communications and Terminal Applications Guide, SC34-1705 (or Communications and Terminal Applications Guide)
- IBM Series/1 Event Driven Executive Commercial Applications Development Guide, SC34-0381 (or Commercial Applications Development Guide)
- IBM Series/1 Event Driven Executive Program Preparation Guide, SC34-1704 (or Program Preparation Guide)
- IBM Series/1 Event Driven Executive Messages and Codes, SC34-0403 (or Messages and Codes)
- IBM Series/1 Event Driven Executive Systems Network Architecture and Remote Job Entry Guide, SC34-0402 (or Systems Network Architecture and Remote Job Entry Guide)
- IBM Series/1 Event Driven Executive Indexed Access Method Version 2 Guide, SC34-0404 (or Indexed Access Method Version 2 Guide)
- IBM Series/1 Event Driven Executive Internal Design, LY34-0212 (or Internal Design)

- _IBM Series/1 Event Driven Executive System/370 Channel Attach Program Internal Design_, LY34-0215 (or _System/370 Channel Attach Program Internal Design_),

- _IBM Series/1 Event Driven Executive Multiple Terminal Manager Internal Design_, LY34-0190 (or _Multiple Terminal Manager Internal Design_),

- _IBM Series/1 Event Driven Executive Indexed Access Method Internal Design_, LY34-0189 (or _Indexed Access Method Internal Design_),

- _IBM Series/1 Event Driven Executive Indexed Access Method Version 2 Internal Design_, LY24-0238 (or _Indexed Access Method Version 2 Internal Design_)

- _IBM Series/1 Event Driven Executive Systems Network Architecture Internal Design_, LY34-0615 (or _Systems Network Architecture Internal Design_)

- _IBM Series/1 Event Driven Executive Reference Summary_, SX34-0038 (or _Reference Summary_),

SUMMARY OF LIBRARY

System Guide

The System Guide has four parts: Concepts and Facilities, System Generation, Problem Determination, and Data Management.

The concepts and facilities part describes the Event Driven Executive system and its related products and introduces the capabilities of these products. The system generation part presents step-by-step procedures for generating a supervisor tailored to your Series/1 hardware configuration and programming needs. The problem determination part explains how to use Event Driven Executive debugging aids. Data management explains direct access data sets, tape data sets, partitioned data sets, dynamic data set allocation, and sensor I/O.

Operator's Reference

Operator's Reference describes:

- Event Driven Executive utility programs
- Operator commands
- Procedures to execute system and application programs
- The session manager -- a menu-driven interface program that will invoke the programs required for program development

The operator commands and utilities are presented in alphabetic order and include detailed syntax and explanations. Also included is a description of the facilities of the session manager.

Utilities are described either in detail or with a reference to the publication where a more detailed description can be found.

Language Reference

The Language Reference familiarizes you with the Event Driven Language by first grouping the instructions into functional categories. Then the instructions are presented alphabetically, with complete syntax and an explanation of each operand. Instructions that support special functions such as 370 Channel Attach are documented in the manual that describes the function. An appendix summarizes the syntax of all instructions.

The final section of the Language Reference contains examples of using the Event Driven Language to code functions such as:

- Program loading
- User exit routine

- I/O level control program
- Indexing and hardware register usage

Communications and Terminal Applications Guide

The Communications and Terminal Applications Guide introduces the Event Driven Executive communications and terminal support.

The Communications and Terminal Applications Guide describes:

- Binary synchronous communications
- Asynchronous communications
- The Host Communications Facility
- Series/1 to Series/1 Attachment support
- The Remote Management Utility
- The System/370 Channel Attach Program
- General Purpose Interface Bus support.

The Communications and Terminal Applications Guide contains the utilities and coding details on Event Driven Language instructions needed for communications support and advanced terminal applications such as graphics.

Commercial Applications Development Guide

The Commercial Applications Development Guide describes:

- The Indexed Access Method Version 1
- The Multiple Terminal Manager
- Printer spooling
- High Level Language interface to static screen subroutines

The description of the Indexed Access Method explains how to plan, design, and build indexed data sets. Also included is information on how to write applications that use indexed data sets.

The Multiple Terminal Manager section explains how to write applications to run under the Multiple Terminal Manager and how to use the Multiple Terminal Manager.

The description of printer spooling explains how to install and use the spooling facility, and describes the spool utility and spool operator commands.

Program Preparation Guide

The Program Preparation Guide describes:

- The Event Driven Language compiler
- The Series/1 assembler
- The host assembler
- The linkage editor
- The object program converter (native and host)
- The data set and overlay prefind utility

SNA and Remote Job Entry Guide

Systems Network Architecture and Remote Job Entry Guide describes the Systems Network Architecture services and functions that are available through a high-level instruction set, and explains Systems Network Architecture formats and protocols. The manual also describes how to use the Systems Network Architecture Remote Job Entry utility.

Messages and Codes

Messages and Codes describes many of the informational, warning, and error messages produced by the Event Driven Executive system and its related products. The manual also contains return codes, completion codes, and post codes issued by these products.

Indexed Access Method Version 2 Guide

This manual describes Version 2 of the Indexed Access Method. It explains how to set up and access an indexed file, and how to use the Indexed Access Method utilities. The manual also contains examples of using the Indexed Access Method in Event Driven Language, COBOL, and PL/I programs.

Internal Designs

Internal Design describes the internal logic flow and specifications of the Event Driven Executive system so that you can understand how the system interfaces with application programs. It familiarizes you with the design and implementation by describing the purpose, function, and operation of the various Event Driven Executive system programs.

Multiple Terminal Manager Internal Design, System/370 Channel Attach Internal Design, Indexed Access Method Internal Design, Indexed Access Method Version 2 Internal Design, and Systems Network Architecture Internal Design describe the internal logic flow and specifications of these programs.

Unlike the other manuals in the library, the Internal Design books contain material that is the licensed property of IBM and they are available only to licensed users of the products that they describe.

Reference Summary

The Reference Summary is a pocket-sized booklet to be used for quick reference. It lists the Event Driven Language instructions with their syntax, the utility and program preparation commands, and completion, post, and return codes.

OTHER EVENT DRIVEN EXECUTIVE PROGRAMMING PUBLICATIONS

The following is a list of Event Driven Executive publications that you may need to code application programs in languages other than the Event Driven Language.

- IBM Series/1 FORTRAN IV Language Reference, GC34-0133.
- IBM Series/1 Event Driven Executive FORTRAN IV User's Guide, SC34-0315.
- IBM Series/1 Mathematical and Functional Subroutine Library User's Guide, SC34-0139.
- IBM Series/1 Event Driven Executive PL/I Language Reference, GC34-0147.
- IBM Series/1 Event Driven Executive PL/I User's Guide, SC34-0148.
- IBM Series/1 COBOL Language Reference, GC34-0234.
- IBM Series/1 COBOL Version 2 Language Reference, SC34-0392
- IBM Series/1 Event Driven Executive COBOL Programmer's Guide, SL23-0014.
- IBM Series/1 Event Driven Executive COBOL Version 2 Programmer's Guide, SC34-0393
- IBM Series/1 Event Driven Executive Macro Assembler Reference,GC34-0317.
- IBM Series/1 Macro Assembler Reference Summary, SX34-0128

OTHER PROGRAMMING PUBLICATIONS

The following publications may be required for reference.

- IBM Series/1 Programming System Summary, GC34-0285.
- IBM Series/1 Event Driven Executive Study Guide, SR30-0436.
- IBM Data Processing Glossary, GC20-1699.
- DOS/VS Tape Labels Manual, GC33-5374
- IBM Series/1 Event Driven Executive Sort/Merge Programmer's Guide, SL23-0016
- IBM Series/1 Data Collection Interactive Programming RPQ P82600 User's Guide, SC34-1654.
- IBM Series/1 Host Communications Facility Program Description Manual, SH20-1819.
- IBM OS/VS Basic Telecommunications Access Method (BTAM), GC27-6980.
- General Information — Binary Synchronous Communications, GA27-3004.
- IBM System/370 Program Preparation Facility, SB30-1072.

SERIES/1 SYSTEM LIBRARY PUBLICATIONS

The following publications contain hardware-related information.

- IBM Series/1 4952 Processor and Processor Features Description, GA34-0084.
- IBM Series/1 4953 Processor and Processor Features Description, GA34-0022.
- IBM Series/1 4955 Processor and Processor Features Description, GA34-0021.
- IBM Series/1 Communications Features Description, GA34-0028.
- IBM 3101 Display Terminal Description, GA18-2033.
- IBM Series/1 4962 Disk Storage Unit and 4964 Diskette Unit Description, GA34-0024.
- IBM Series/1 4963 Disk Subsystem Description, GA34-0051.
- IBM Series/1 4966 Diskette Magazine Unit Description, GA34-0052.
- IBM Series/1 4969 Magnetic Tape Subsystem Description, GA34-0087.

- IBM Series/1 4973 Line Printer Description, GA34-0044.

- IBM Series/1 4974 Printer Description, GA34-0025.

- IBM Series/1 4978-1 Display Station (RPQ D02055) and Attachment (RPQ D02038) General Information, GA34-1550

- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02056) General Information, GA34-1551

- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02057) General Information, GA34-1552

- IBM Series/1 4978-1 Display Station Keyboards (RPQ D02064 and D02065) General Information, GA34-1553

- IBM Series/1 4979 Display Station Description, GA34-0026

- IBM Series/1 4982 Sensor Input/Output Unit Description, GA34-0027

- IBM Diskette - General Information Manual, GA21-9182

- IBM Series/1 General Purpose Interface Bus (GPIB) Adapter - RPQ D02118, GA34-1556

- IBM Series/1 to Series/1 Attachment RPQs D02241 & D02242 Custom Feature, GA34-1561

- IBM Series/1 System/370 Channel Attachment Feature and 4993 Model 1 Series/1 System/370 Termination Enclosure Description, GA34-0057

- IBM Series/1 Data Collection Interactive RPQs D02312, D02313, and D02314 Custom Feature, GA34-1567

GLOSSARY OF TERMS AND ABBREVIATIONS

This glossary contains terms and abbreviations used in the Series/1 Event Driven Executive software publications. All software and hardware terms are Series/1 oriented. This glossary also serves as a supplement to the IBM Data Processing Glossary (GC20-1699).

$SYSLOGA, $SYSLOGB: The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG: The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR: The name of the system printer.

ACCA: See asynchronous communications control adapter.

address key: Identifies a set of Series/1 segmentation registers and represents an address space. It is one less than the partition number.

address space: The logical storage identified by an address key. An address space is the storage for a partition.

application program manager: The component of the Multiple Terminal Manager that provides the program management facilities required to process user requests. It controls the contents of a program area and the execution of programs within the area.

application program stub: A collection of subroutines that are appended to a program by the linkage editor to provide the link from the application program to the Multiple Terminal Manager facilities.

asynchronous communications control adapter: An ASCII terminal attached via #1610, #2091 with #2092, or #2095 with #2096 adapters.

attention list: A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup: A copy of data to be used in the event the original data is lost or damaged.

base record slots: Space in an indexed file that is reserved for based records to be placed.

base records: Records are placed into an indexed file while in load mode or inserted in process mode with a new high key.

basic exchange format: A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB): A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

block: (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

block mode: The transmission mode in which the 3101 Display Station transmits a data data stream, which has been edited and stored, when the SEND key is pressed.

BSCDDB: See binary synchronous device data block.

buffer: An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing: Access of a tape without any label processing support.

CCB: See terminal control block.

central buffer: The buffer used by the Indexed Access Method for all transfers of information between main storage and indexed files.

character image: An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table: An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

character mode: The transmission mode in which the 3101 Display Station immediately sends a character when a keyboard key is pressed.

cluster: In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction): A character used with ACCA terminal to indicate a reverse in the direction of data movement.

cold start: Starting the spool facility by erasing any spooled jobs remaining in the spool data set from any previous spool session.

command: A character string from a source external to the system that represents a request for action by the system.

common area: A user-defined data area that is mapped into every partition at the same address. It can be used to contain control blocks or data that will be accessed by more than one program.

completion code: An indicator that reflects the status of the execution of a program. The completion code is displayed or printed on the program's output device.

controller: A device that has the capability of configuring the GPIB bus by designating which devices are active, which devices are listeners, and which device is the talker. In Series/1 GPIB implementation, the Series/1 is always the controller.

conversion: See update.

cross partition service: A function that accesses data in two partitions.

data block: In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data record: In an indexed file, the records containing customer data.

data set: A group of records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB): A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down: An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE: See directory control entry.

DDB: See disk data block.

direct access: (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation. (3) A condition in terminal I/O where a READTEXT or a PRINTEXT is directed to a buffer which was previously enqueued upon by an IOCB.

directory: (1) A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free spaces. (2) For the Indexed Access Method Version 2, a data set that defines the relationship between primary and secondary indexed files (secondary index support).

directory control entry (DCE): The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME): A 32-byte directory entry describing an allocated data set or volume.

disk data block (DDB): A control block that describes a direct access volume.

display station: An IBM 4978, 4979, or 3101 display terminal or similar terminal with a keyboard and a video display.

DME: See directory member entry.

DSCB: See data set control block.

dynamic storage: An increment of storage that is appended to a program when it is loaded.

end-of-data indicator: A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB: See event control block.

EDL: See Event Driven Language.

emulator: The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT): A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

event control block (ECB): A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

event driven language (EDL): The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output): An EDL facility that provides user controlled access to Series/1 input/output devices.

external label: A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN): The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA: See file control area.

FCB: See file control block.

file: A set of related records treated as a logical unit. Although file is often used interchangeably with data set, it usually refers to an indexed or a sequential data set.

file control area (FCA): A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB): The first block of an indexed file. It contains descriptive information about the data contained in the file.

file control block extension: The second block of an indexed file. It contains the file definition parameters used to define the file.

file manager: A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

formatted screen image: A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool: In an indexed data set, a group of blocks that can be used for either data blocks or index blocks. These differ from other free blocks in that these are not initially assigned to specific logical positions in the file.

free space: In an indexed file, records blocks that do not currently contain data, and are available for use.

free space entry (FSE): An 8-byte directory entry defining an area of free space within a volume or a device.

FSE: See free space entry.

general purpose interface bus: The IEEE Standard 488-1975 that allows various interconnected devices to be attached to the GPIB adapter (RPQ D02118).

GPIB: See general purpose interface bus.

group: A unit of 100 records in the spool data set allocated to a spool job.

H exchange format: A standard format for exchanging data on diskettes between systems or devices.

hardware timer: The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card or the native timer (4952 only). Only one or the other is supported by the Event Driven Executive.

host assembler: The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system: Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB: See indexed access control block.

IAR: See instruction address register.

ICB: See indexed access control block.

IIB: See interrupt information byte.

image store: The area in a 4978 that contains the character image table.

immediate data: A self-defining term used as the operand of an instruction. It consists of numbers, messages or values which are processed directly by the computer and which do not serve as addresses or pointers to other data in storage.

index: In an indexed file, an ordered collection of pairs of keys and pointers, used to sequence and locate records.

index block: In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB): The control block that relates an application program to an indexed file.

indexed access method: An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set: Synonym for indexed file.

indexed file: A file specifically created, formatted and used by the Indexed Access Method. An indexed file is sometimes called an indexed data set.

index entry: In an indexed file, a key-pointer pair, where the pointer is used to locate a lower-level index block or a data block.

index register (#1, #2): Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer: (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB): A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer, or an optional reference to a user provided buffer.

instruction address register (IAR): The pointer that identifies the instruction currently being executed. The Series/1 maintains a hardware IAR to determine the Series/1 assembler instruction being executed. It is located in the level status block (LSB).

interactive: The mode in which a program conducts a continuous dialogue between the user and the system.

internal label: An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB): In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

invoke: To load and activate a program, utility, procedure, or subroutine.

job: A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement: A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor: The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

jumper: A wire or pair of wires which are used for the arbitrary connection between two circuits or pins in an attachment card.

key: In the Indexed Access Method, one or more consecutive characters used to identify a record and establish its order with respect to other records. See also key field. See also key field.

key field: A field, located in the same position in each record of an indexed file, whose content is used for the key of a record.

level status block (LSB): A Series/1 hardware data area that contains processor status.

library: A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line: A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit: The process of resolving external symbols in one or more object modules. A link edit can be performed with $EDXLINK, whose output is a loadable program, or with $LINK, whose output is a single, composite object module that is the input to the update process.

listener: A controller or active device on a GPIB bus that is configured to accept information from the bus.

load mode: In the Indexed Access Method, the mode in which records are loaded into base record slots in an indexed file.

load module: A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point: (1) Address in the partition where a program is loaded. (2) A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock: In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

logical screen: A screen defined by margin settings, such as the TOPM, BOTM, LEFTM and RIGHTM parameters of the TERMINAL or IOCB statement.

LSB: See level status block.

member: A term used to identify a named portion of a partitioned data set (PDS). Sometimes member is also used as a synonym for a data set. See data set.

menu: A formatted screen image containing a list of options. The user selects an option to invoke a program.

menu-driven: The mode of processing in which input consists of the responses to prompting from an option menu.

multifile volume: A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager: An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file: A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

new high key: A key higher than any other key in an indexed file.

non-labeled tapes: Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character: A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu: A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer: (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay: The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program or by calling overlay segments that have been specified in the OVERLAY statement of $EDXLINK.

overlay area: A storage area within a program reserved for overlay programs specified in the PROGRAM statement or overlay segments specified in the OVERLAY statement in $EDXLINK.

overlay program: A program in which certain control sections can use the same storage location at different times during execution. An overlay program can execute concurrently as an asynchronous task with other programs and is specified in the EDL PROGRAM statement in the main program.

overlay segment: A self-contained portion of a program that is called and sequentially executes as a synchronous task. The entire program that calls the overlay segment need not be maintained in storage while the overlay segment is executing. An overlay segment is specified in the OVERLAY statement of $EDXLINK.

parameter selection menu: A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition: A contiguous fixed-sized area of storage. Each partition is a separate address space.

physical timer: Synonym for hardware timer.

post code: An indicator returned to the event control block to indicate the completion of a wait or an event.

prefind: To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary file: An indexed file containing the data records and primary index.

primary file entry: For the Indexed Access Method Version 2, an entry in the directory describing a primary file.

primary index: The index portion of a primary file. This is used to access data records when the primary key is specified.

primary key: In an indexed file, the key used to uniquely identify a data record.

primary-level index block: In an indexed file, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu: The program selection screen displayed by the Multiple Terminal Manager.

primary option menu: The first full screen display provided by the Session Manager.

primary task: The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority: A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode: In the Indexed Access Method, the mode in which records can be retrieved, updated, inserted or deleted.

processor status word (PSW): A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program: A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header: The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager: A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field: A field in which the operator cannot use the keyboard to enter, modify, or erase data.

PSW: See processor status word.

QCB: See queue control block.

QD: See queue descriptor.

QE: See queue element.

queue control block (QCB): A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD): A control block describing a queue built by the DEFINEQ instruction.

queue element (QE): An entry in the queue defined by the queue descriptor.

record: (1) The smallest unit of direct access storage that can be accessed by an application program on a disk or diskette using READ and WRITE. Records are 256 bytes in length. (2) In the Indexed Access Method, the logical unit that is transferred between $IAM and the user's buffer. The length of the buffer is defined by the user.

recovery: The use of backup data to recreate data that has been lost or damaged.

reflective marker: A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative record number: An integer value identifying the position of a record in a data set relative to the beginning of the data set. The first record of a data set is record one, the second is record two, the third is record three.

reorganize: The process of copying the data in an indexed file to another indexed file in a manner that rearranges the data for more optimum processing and free space distribution.

restart: Starting the spool facility by processing spool jobs remaining in the spool data set from any previous spool session.

return code: An indicator that reflects the results of the execution of an instruction or subroutine. The return code is placed in the task code word (at the beginning of the task control block).

roll screen: A display screen which is logically segmented into an optional history area and a work area. Output directed to the screen starts display at the beginning of the work area and continues on down in a line-by-line sequence. When the work area gets full, the operator presses ENTER/SEND and its contents are shifted into the optional history area and the work area itself is erased. Output now starts again at the beginning of the work area.

SBIOCB: See sensor based I/O control block.

second-level index block: In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary file.: See secondary index.

secondary index: For the Indexed Access Method Version 2, an indexed file used to access data records by their secondary keys. Sometimes called a secondary file.

secondary index entry: For the Indexed Access Method Version 2, this an an entry in the directory describing a secondary index.

secondary key: For the Indexed Access Method Version 2, the key used to uniquely identify a data record.

secondary option menu: In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task: Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

sector: The smallest addressable unit of storage on a disk or diskette. A sector on a 4962 or 4963 disk is equivalent to an Event Driven Executive record. On a 4964 or 4966 diskette, two sectors are equivalent to an Event Driven Executive record.

sensor based I/O control block (SBIOCB): A control block containing information related to sensor I/O operations.

sequential access: The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR): A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

service request: A device generated signal used to inform the GPIB controller that service is required by the issuing device.

session manager: A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource: A resource that can be used by more than one task at the same time.

shut down: See data set shut down.

source module/program: A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

spool job: The set of print records generated by a program (including any overlays) while enqueued to a printer designated as a spool device.

spool session: An invocation and termination of the spool facility.

SRQ: See service request.

standard labels: Fixed length 80-character records on tape containing specific fields of information (a volume label identifying the tape volume, a header label preceding the data records, and a trailer label following the data records).

static screen: A display screen formatted with predetermined protected and unprotected areas. Areas defined as operator prompts or input field names are protected to prevent accidental overlay by input data. Areas defined as input areas are not protected and are usually filled in by an operator. The entire screen is treated as a page of information.

subroutine: A sequence of instructions that may be accessed from one or more points in a program.

supervisor: The component of the Event Driven Executive capable of controlling execution of both system and application programs.

system configuration: The process of defining devices and features attached to the Series/1.

SYSGEN: See system generation.

system generation: The processing of user selected options to create a supervisor tailored to the needs of a specific Series/1 configuration.

system partition: The partition that contains the supervisor (partition number 1, address space 0).

talker: A controller or active device on a GPIB bus that is configured to be the source of information (the sender) on the bus.

tapemark: A control character recorded on tape used to separate files.

task: The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task are secondary tasks.

task code word: The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB): A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor: The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB: See task control block.

terminal: A physical device defined to the EDX system using the TERMINAL configuration statement. EDX terminals include directly attached IBM displays, printers and devices that communicate with the Series/1 in an asynchronous manner.

terminal control block (CCB): A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB): A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager: The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server: A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

terminal support: The support provided by EDX to manage and control terminals. See terminal.

trace range: A specified number of instruction addresses within which the flow of execution can be traced.

transaction oriented applications: Program execution driven by operator actions, such as responses to prompts from the system. Specifically, applications executed under control of the Multiple Terminal Manager.

transaction program: See transaction-oriented applications.

transaction selection menu: A Multiple Terminal Manager display screen (menu) offering the user a choice of functions, such as reading from a data file, displaying data on a terminal, or waiting for a response. Based upon the choice of option, the application program performs the requested processing operation.

unprotected field: A field in which the operator can use the keyboard to enter, modify or erase data. Also called non-protected field.

update: (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit: (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a user written routine can be given control.

vary offline: (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is unknown by the system.

vary online: To place a device in a state where it is available for use by the system.

volume: A disk, diskette, or tape subdivision defined using $INITDSK or $TAPEUT1.

volume label: A label that uniquely identifies a single unit of storage media.

COMMON INDEX

This index is common to the Event Driven Executive library. It is a combination of entries from the publications listed below. Each entry references all of the publications that contain information about the entry, so this index provides a cross-reference between the publications.

Each page number entry contains a single or double letter prefix which identifies the publication where the subject can be found. The letter prefixes have the following meanings:

- AM = Indexed Access Method Internal Design (Version 1)
- CA = Commercial Applications Development Guide
- CT = Communications and Terminal Application Guide
- ID = Internal Design
- IG = Indexed Access Method Version 2 Guide
- IM = Indexed Access Method Version 2 Internal Design
- LR = Language Reference
- M = Multiple Terminal Manager Internal Design
- MC = Messages and Codes
- OR = Operator's Reference
- PP = Program Preparation Guide
- SG = System Guide
- SI = SNA Internal Design
- SM = Sort/Merge Programmer's Guide
- SN = SNA and Remote Job Entry Guide
- T = S/370 Channel Attach Internal Design

Special Characters

READER'S COMMENT FORM

SC34-0404-0

IBM Series/1 Event Driven Executive
Indexed Access Method Version 2 Guide

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Please indicate your name and address in the space below if you wish a reply.

Thank you for your coöperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

Cut or Fold Along Line

Reader's Comment Form

Cut Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape Please Do Not Staple Fold and tape

IBM®

SC34-0404-0
Printed in U.S.A.

SC34-0404-0

Printed in U.S.A.