IBM

Series/1

GC34-0328-0

File No. S1-34

LICENSED
MATERIAL

IBM Series/1

Event Driven Executive Version 1.1

Planning Guide

Program Numbers: 5719-XS1
5719-UT3
5719-XX2
5719-LM5
5719-LM2
5719-MS1 (Multiple Terminal Manager)
5719-AM3 (Indexed Access Method)

IBM

Series/1

GC34-0328-0

File No. S1-34

LICENSED
MATERIAL

IBM Series/1
Event Driven Executive Version 1.1
Planning Guide

Program Numbers: 5719-XS1
5719-UT3
5719-XX2
5719-LM5
5719-LM2
5719-MS1 (Multiple Terminal Manager)
5719-AM3 (Indexed Access Method)

First Edition (July 1979)

Use this publication only for the purpose stated in the Preface.

Changes are periodically made to the information herein; before using this publication in connection with the operation of IBM systems, refer to the latest IBM Series/1 Graphic Bibliography, GA34-0055, for the editions that are applicable and current.

Publications are not stocked at the address given below. Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

This publication could contain technical inaccuracies or typographical errors. A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432. 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.

This publication is to be used for planning purposes only. Any information contained herein is subject to change. For final information concerning any topic discussed in this publication, consult the Version 1.1 manuals listed below when they become available.

This publication provides the information required to design applications that use Version 1.1 of the Event Driven Executive, the Multiple Terminal Manager, and the Indexed Access Method.

HOW THIS MANUAL IS ORGANIZED

- Part 1 describes Version 1.1 of the Event Driven Executive and points out the enhancements it provides over Version 1.0.
- Part 2 describes the Multiple Terminal Manager, which you can use to simplify the definition of transaction-oriented applications.
- Part 3 describes the Indexed Access Method, which you can use to access and maintain your data in a keyed (indexed) organization.

Before reading this manual, you should be familiar with the Event Driven Executive. See the list below for publications that can help you.

RELATED PUBLICATIONS:

The following publications are now available for Version 1.0 of the Event Driven Executive. The Version 1.1 publications will be released in October, 1979, under the same numbers and titles (except as noted).

- IBM Series/1 System Summary, GC34-0285.
- IBM Series/1 Event Driven Executive System Guide, SC34-0312.
- IBM Series/1 Event Driven Executive Utilities, Operator Commands, and Program Preparation, SC34-0313.

  Note. The Version 1.1 release of this document will be titled: IBM Series/1 Event Driven Executive Utilities, Commands, Program Preparation and Messages and Codes, SC34-0313.
- IBM Series/1 Event Driven Executive Language Reference, SC34-0314.
- IBM Series/1 Event Driven Executive Macro Assembler, SC34-0317.

- IBM Series/1 Event Driven Executive Communications Guide, SC34-0316.

  Note. The Version 1.1 release of this document will be: IBM Series/1 Event Driven Executive Communications and Terminal Applications Guide, SC34-0316.

INTRODUCTION

Version 1.1 of the Event Driven Executive, available in October 1979, expands the scope of the data processing solutions possible with the Event Driven Executive. Improvements include:

- Support for 128KB of storage on the 4952
- Applications programs up to 64KB
- Cross-partition communication capability
- Diagnostic and recovery improvements

The changes that make this possible are described in this planning guide.

Version 1.1 of the Event Driven Executive system includes service releases of each of the following licensed programs:

- Basic Supervisor and Emulator (5719-XS1)
- Utilities (5719-UT3)
- Program Preparation Facility (5719-XX2)
- Macro Assembler (5719-ASA)
- Macro Library (5719-LM5)
- Macro Library/Host (5740-LM2)
- COBOL Compiler and Resident Library (5719-CB3)
- COBOL Transient Library (5719-CB4)
- FORTRAN IV Compiler and Object Library Version 2 (5719-F02)
- Sort/Merge (5719-SM2)
- Mathematical and Functional Subroutine Library (5719-LM3)

The following new licensed programs have also been announced for use with Version 1.1 of the system and will be available in October 1979.

- Indexed Access Method (5719-AM3)
- Multiple Terminal Manager (5719-MS1)

These new programs are also described in this planning guide.

ADDRESS TRANSLATOR SUPPORT

VERSION 1.0

The Event Driven Executive (EDX) operating system for the Series/1 currently supports the address translator feature of the 4952 and 4955 processors. The programs executing in each storage partition have direct access to the functions and data areas in the supervisor, in much the same way as a program executing in a non-address translated, single partition system. Although this mechanism offers the advantages of conceptual simplicity and compatibility with earlier versions of the operating system, it has two drawbacks. First, segmentation registers are used to map the system area into each partition defined by the user. In systems with processor storage approaching 128K bytes and two sets of segment registers, the result may be an inadequate number of segmentation registers to map the entire storage area. Thus, processor storage can exist which cannot be accessed by either user or system tasks. Second, the user application may be size-constrained by the necessity to leave 20-32KB or more for the system area in each 64KB partition.

The system requires that two facilities be available to user programs and system utilities regardless of the addressing mechanism used. These are the ability to invoke supervisor functions through Series/1 assembler language instructions or the Event Driven Language and the ability to reference system data areas, again in either a direct or indirect fashion. As noted above, the current version of EDX provides these functions by logically mapping the supervisor into each partition.

VERSION 1.1

Version 1.1 of EDX provides a supervisor that resides only in partition 1, thus making it possible to support application programs up to 64K bytes in length in the other partitions. Elimination of redundant mappings of the sys-

tem area frees more of the processor's segmentation registers for use in addressing unique physical storage, thus increasing the amount of storage that can be effectively put to use on processors such as the 4952. The ability to optionally map a portion of the system area into each partition provides the capability for shared storage areas (such as $SYSCOM) without wasting large amounts of application space on other control program modules as well. Figure 1 illustrates some of the storage configurations now possible with Version 1.1.

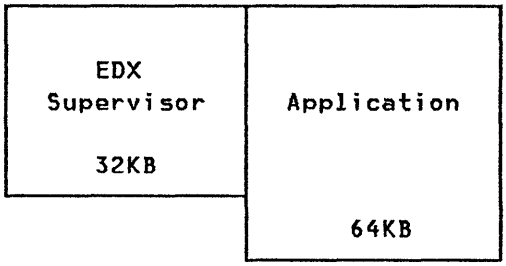
Another advantage of the new storage map is that it provides increased protection of the system area from accidental modification by applications. Previously, a program error could accidentally damage the system and affect all programs and partitions. Because the Version 1.1 system area resides in partition 1 only, the chance of accidental modification by programs in other partitions is considerably reduced.

To complement the storage map improvement the enhanced address translator support includes several extensions to system functions for the multi-partition environment. These extensions make it possible to move data and signal events and manage resources across partition boundaries, thus making multiple partition applications easier to design and implement. These facilities are described in more detail under the heading "Multi-Partition Extensions."

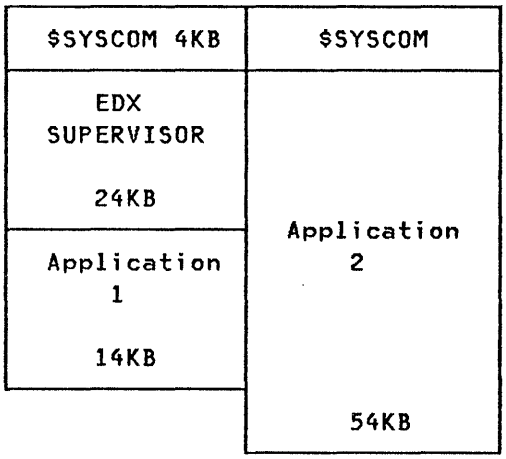
Version 1.1 also contains improved diagnostic and recovery capability. Optional facilities include I/O error logging, a task error recovery exit, a program exception trace, and a storage dump. These facilities are described in more detail under the heading "Diagnostic and Recovery Improvements."

COMPATIBILITY

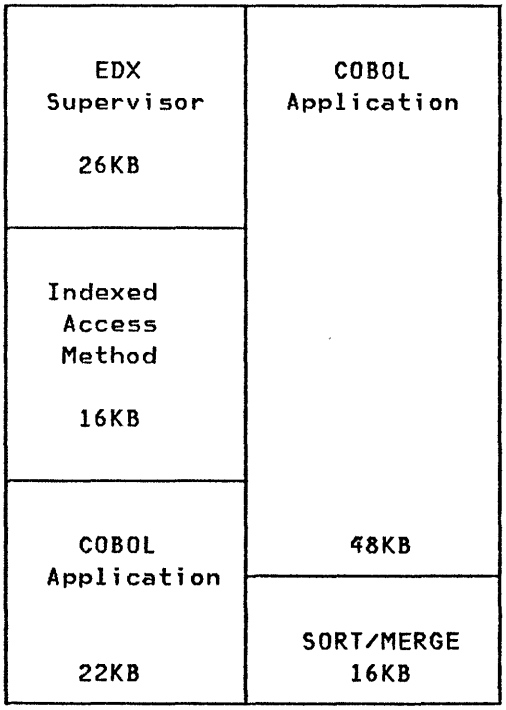
Version 1.1 preserves all the standard Event Driven Executive application program interfaces and as many of the application implementation techniques as possible. Version 1.1 is compatible with Version 1.0 in terms of source programs, data sets and functional content. Each of these is explained more fully below. In general, you may expect that applications written to run on Version 1.0 will run on Version 1.1 after recompilation. Since most applications written for the Event Driven Executive FDP Version 2 would also run on Version 1.0 of the licensed program, the information in this guide is also applicable to them.



- 96KB 4955
- Large single application



- 96KB 4952
- 2 application spaces
- Shared ($SYSCOM) area



- 128KB 4952 or 4955
- Large COBOL application using Sort and Indexed Access Method
- Second COBOL application also utilizing Indexed Access Method
- Indexed Access Method which serves both COBOL applications plus - Sort/Merge (buffer space expanded by 2K bytes)

Figure 1. Possible Storage Configurations

SOURCE PROGRAM COMPATIBILITY

COBOL and FORTRAN programs are completely source compatible and require only that they be recompiled using Version 1.1 of the COBOL or FORTRAN compiler. An additional module of less than 200 bytes will be automatically included (by Autocall) during link-edit. Execution of COBOL programs will require Version 1.1 of the COBOL Transient Library. Version 1.1 of COBOL and FORTRAN will be available concurrent with Version 1.1 of supervisor and will be automatically shipped to existing licensees.

Event Driven Language (EDL) and Series/1 Assembler Language programs that use standard application interfaces are source compatible. Refer to the topic below titled "Application Program Interfaces" for a list of source compatible interfaces. However, any programs with explicit dependencies on access to the system area may:

1. Be required to execute in partition 1 only so that unrestricted access to the system area is possible.

2. Require that the option to map some or all of the system area into each partition be used to provide the ability to reference system area control blocks.

3. Require source modification to utilize the cross address space capabilities added to various functions in Version 1.1.

Recompilation of the Event Driven Language or assembler language programs requires Version 1.1 of the appropriate program preparation vehicle. All these releases will be available concurrent with Version 1.1 of the Supervisor and Emulator and will be automatically shipped to existing licensees. The following program preparation vehicles may be used:

1. Program Preparation ($EDXASM) Version 1.1 (for programs consisting exclusively of Event Driven Language statements).

2. Macro Library Version 1.1 and Macro Assembler Version 1.1. If it is desired to compile programs on Version 1.0 for later execution on a Version 1.1 system, the Version 1.1 Macro Library can also be used with the Version 1.0 Macro Assembler.

3. Macro Library/Host Version 1.1 and any current release of the Series/1 370 Host Assembler.

Application programs that contain Series/1 assembler language instructions will require 1-3 additional modules to be link-edited with the application. These modules are

provided with the Version 1.1 supervisor for the purpose of handling the special interface used by assembler language routines under EDX. Most assembler language applications will require only the basic RETURN interface module (about 40 bytes) but some applications may require additional modules that support the special interfaces SVC, SETBUSY and SUPEXIT. The interface modules total approximately 200 bytes and can be automatically included where needed with the AUTOCALL facility of $LINK if EXTRN statements for the desired modules are included with the program. See the "Special Considerations" topic for more information.

APPLICATION PROGRAM INTERFACES

The primary application program interface is documented in the Event Driven Executive Language Reference manual. Users of COBOL or FORTRAN may also view their respective language reference manuals and users guides as their interface. In addition to these, certain other interfaces and data areas are available for reference and use by application programs. These are the following:

- The Branch interfaces named RETURN, SETBUSY, SUPEXIT and SVC and their respective parameter lists.
- The CALL interface for DSOPEN, including its parameters and data area.
- The CALL interface for SETEOD including its parameter list and data area.
- The following interfaces as described in the Communications Guide: BSCOPEN, BSCCLOSE, BSCREAD, BSCWRITE, BSCIOB, and TP (all functions).
- The first two words of the Task Control Block (TCB) system data area.
- The first two words of the Data Set Control Block (DSCB).

Programs should use only the above interfaces so that they will be source-statement compatible across releases of EDX.

DATA SET COMPATIBILITY

Data sets created or updated using Version 1.0 can be used on Version 1.1 without change.

FUNCTIONAL CONTENT COMPATIBILITY

All the functions defined in the Language Reference Manual, the Utilities Operator Commands and Program Preparation and the Communications Guide for Version 1.0 are also supported by Version 1.1.

STORAGE SIZES

Version 1.1 of the Supervisor and Emulator is slightly larger than Version 1.0. For planning purposes you may assume that the system area of identical configurations will increase by approximately 5% for a configuration with 64KB or less and by approximately 10% for configurations with more than 64KB of storage. Actual increases will depend on the particular functions included in each system.

Systems installed on a 4952 processor with 128KB of storage will gain the use of up to 32KB of additional storage in return for the 2-4KB increase in control program size. Users of the 4955 processor with greater than 64KB will also find that the increase is offset by application improvements made possible by the availability of larger application spaces and the new multi-partition extensions. Potential benefits include the ability to avoid the need for complex overlays or applications that must be split apart so they can be executed in separate partitions, and the ability to keep more data in storage instead of spilling it to disk to save space.

Changes in program size are expected to be minimal for most applications written in the Event Driven Language. A straightforward program with a single task and no attention lists (ATTNLIST) will increase about 50 bytes. Each attention list will expand about 120 bytes (see "Special Considerations" for more information on ATTNLIST). Programs containing Series/1 assembler language code (including all COBOL and FORTRAN programs) will require an additional 50 to 200 bytes because of the inclusion of the interface modules described in the section on "Source Program Compatibility."

MULTI-PARTITION EXTENSIONS

As part of the changes in the address translator support several system functions have been altered to accommodate the new environment. These changes remove many of the restrictions of the Version 1.0 address translator and allow applications written in the Event Driven Language or assembler language to take maximum advantage of the multi-partition environment. With the Version 1.1 address translator support it is possible to:

- Move data directly from the executing program's partition to another, or vice versa, using the MOVE instruction.
- Enqueue or dequeue (ENQ/DEQ) on a resource (QCB) in another partition without the use of a common area ($SYSCOM).
- LOAD a program in another partition and pass parameters to it or wait for its completion.
- READ or WRITE using a buffer in a partition other than the executing program's.
- ATTACH a task in another partition.
- WAIT or POST an event (ECB) in a partition other than the invoker's and without a common area ($SYSCOM).
- Locate a program by name and determine its address and partition. This allows independently loaded programs to find each other at execution time without using a common area ($SYSCOM).

DIAGNOSTIC AND RECOVERY IMPROVEMENTS

I/O ERROR RECORDING

Included in Version 1.1 is the optional ability to log the occurrence of hardware errors on a data set. This facility, which is used by EDX device services and is available to user-written device handlers and EXIO users, provides error data to assist in the detection and correction of hardware errors. A utility to print the contents of the log data set is included.

TASK ERROR EXIT

Application programs can elect to process errors themselves instead of relying on standard system actions. This enables diagnostic and recovery actions to be tailored to each user's application environment. Each task can identify a routine to receive control if a processor malfunction or program exception occurs during the execution of the task. If an error occurs, execution of the task is resumed at the task error exit routine in lieu of the standard system action of terminating the task.

SYSTEM AREA PROTECTION

Isolation of the system area from all but one application partition constrains most application errors to a single partition instead of allowing impact to the entire machine through damage to the system area.

PROGRAM EXCEPTION TRACE

Program exception data can be captured in an in-storage table, providing a chronological history of exceptional conditions for use in debugging.

STORAGE DUMP

A storage dump utility provides the ability to stop on program exceptions and record the contents of the storage in a data set. A second utility prints the dump data set.

SPECIAL CONSIDERATIONS

Certain interfaces and implementation techniques have been affected by the Version 1.1 address translator changes. Application programs that use any of the interfaces or techniques described in this section will require review to determine if changes will be necessary or desirable in the new execution environment provided by Version 1.1. These considerations are in addition to those listed under "Compatibility."

REFERENCES TO SYSTEM AREA DATA

Applications that depend on explicit reference to data (control blocks) in the system area are definitely affected by the new multi-partition environment because the system area is not addressable by the program unless special action is taken. Data areas such as the DDB (disk), CCB (terminal) and CMDTABLE (command Table) are not generally accessible in a multi-partition system. There are three approaches that can be used to move applications with this dependency onto a multiple partition Version 1.1 system.

1. Execute the application in partition 1 only. Since the system area also resides in partition 1, the application will have unrestricted access to it without the need for any changes.

2. Map some or all of the system area into every partition so that the desired data areas are accessible from every partition. This requires generation of a system using a new, optional parameter on the SYSTEM statement, but it requires no changes to the application program.

3. Alter the application to use the new multi-partition functions to obtain the desired data. Some applications may be able to use the new functions exclusively instead of depending on direct use of the system area. This is recommended where possible because the content and format of the system area are subject to change. In addition, it should be possible to eliminate the need for $SYSCOM from many applications.

INVOKING PROGRAMS THAT RESIDE IN THE SYSTEM AREA

Certain programs that reside in the system area may be branch-entered by assembler language application programs. Only the following four entry point names are supported: RETURN, SVC, SETBUSY and SUPEXIT. To use these entry points in a Version 1.1 system it will be necessary to link-edit ($LINK) additional module(s) with the application program. No source code modifications are required. The additional modules, which are provided with the Version 1.1 supervisor and emulator, are about 200 bytes in total size. Only the module(s) required for the particular entry point(s) in the program need be included. Mapping the system area into all partitions does not eliminate the requirement to use the linkage module.

USER-WRITTEN SUPERVISOR EXTENSIONS

In general, user-written extensions to the supervisor will require careful review for dependencies on addressing data in the application's partition. The keyboard task will always execute with address key 0 because it physically resides in partition 1. Change Partition ($CP) will no longer change the processor address key register; instead the new partition value will be placed in a TCB field, TCBADS. System functions must use this field to determine the actual partition in use. The TCBAKR field should not be used for this purpose.

VIRTUAL TERMINALS

Applications that use virtual terminals may require modification if the TERMINAL statement defining the virtual terminal is contained in the application program rather than in the system configuration module ($EDXDEFS). Commencing with Version 1.1, all TERMINAL statements must be in partition 1 and application programs in other partitions must use an IOCB statement to gain access to the virtual terminal. To help provide source compatibility, TERMINAL statements coded in application programs will automatically be converted to IOCB statements during assembly. However, TERMINAL statements with the same names and parameters as those in the application program must be added to the system configuration module ($EDXDEFS).

The following example, which is explained in the "Advanced Topics" section, the Version 1.0 System Guide, illustrates a typical use of virtual terminals:

```
A       TERMINAL DEVICE=VIRT,ADDRESS=B,SYNC=YES

B       TERMINAL DEVICE=VIRT,ADDRESS=A,END=YES

        ENQT B

        LOAD $TERMUT1,LOGMSG=NO,END=ENDWAIT

        ENQT A
```

The result is a virtual channel between $TERMUT1 and the program that loaded it. In Version 1.1 the TERMINAL statements in the example above will actually generate the following two IOCB statements:

```
A       IOCB      A

B       IOCB      B
```

The IOCB statements that replace the application's TERMINAL statements must point to TERMINAL statements in the system configuration module ($EDXDEFS). Therefore, for this sample to work the following two statements must be added to the system configuration module:

```
A       TERMINAL DEVICE=VIRT,ADDRESS=B,SYNC=YES

B       TERMINAL DEVICE=VIRT,ADDRESS=A,END=YES
```

The effect of this change is to increase the size of the system area by about 700 bytes and to decrease the size of the virtual terminal application by about 640 bytes. Since multiple programs can share the virtual terminal definitions (using ENQT) when they are part of the system configuration, this may result in an overall decrease in storage requirements. However, programs that use the same virtual terminal pair can not execute at the same time.

SENSOR I/O APPLICATIONS

Applications using the specialized sensor I/O capabilities of EDX may be affected by changes in timing that occur in Version 1.1. The special process interrupt interface (IODEF SPECPI) is slightly faster than in Version 1.0 if the application is executed in partition 1 but is estimated to be 25-30 microseconds slower (on a 4955 processor) if executed in some other partition. Programs with extreme time sensitivity should therefore use SPECPI TYPE=GROUP and be executed in partition 1.

The method of return from the SPECPI exit to the supervisor may also require modification. For SPECPI TYPE=GROUP there is no change, but for SPECPI TYPE=BIT a new return statement, SPECPIRT, is provided. The Version 1.0 technique, which used register 7, is valid only in partition 1. The new statement, SPECPIRT will work in any partition.

USE OF ATTENTION LISTS (ATTNLIST)

Programs that use ATTNLIST exits to provide new commands or intercept existing system commands may be affected by a change in the method of executing attention routines. In Version 1.0 the ATTNLIST routine was executed directly from the keyboard task for the terminal in use. In Version

1.1, the keyboard task is always executed in partition 1 while the attention exits themselves are executed in whatever partition the exit program resides in. Therefore, the keyboard task will ATTACH a special ATTNLIST TCB to execute the attention exit. This special TCB is automatically created when a program containing an ATTNLIST statement is assembled, increasing program size by approximately 120 bytes. The attention routine is executed in the partition it resides in and therefore does not have direct access to the system area. If the attention exit needs to reference the system area, refer to the section on "References To System Area Data."

PROGRAM (HEADER) OR TCB DEPENDENCIES

The PROGRAM header and TCB's that reside within the users' application program have changed slightly in size and format. Programs that reference or modify these control blocks except for the first two words of a TCB or a DSn (DSCB) may require source code modification. While most fields have changed only in offset within the data area, no assumptions should be made about compatibility. In particular, use of any reserved fields or fields that contain address keys indicates that a change would be required. IBM recommends against any dependency on these data areas.

&SYSCOM USAGE

Applications that use a $SYSCOM shared area should not require change. However, you may wish to control the placement of the $SYSCOM CSECT to minimize the amount of storage shared across partitions. All storage locations from 0 to the boundary specified in a new SYSTEM statement parameter are shared (mapped) by all partitions. Therefore it is usually advisable to include the $SYSCOM CSECT near the beginning of the supervisor load module rather than at the high end.

OVERVIEW

MULTIPLE TERMINAL MANAGER CONCEPTS

The Series/1 Multiple Terminal Manager (5719-MS1) is an application program that operates under the Series/1 Event Driven Executive. It provides a set of high-level functions that simplify the definition of transaction-oriented applications such as inquiry, file update, data collection, and order entry.

Transaction-oriented means that program execution is driven by operator actions, typically, responses to prompts from the system.

This prompt-response-process cycle between the program and the terminal operator is the basic principle for the design of applications using the Multiple Terminal Manager.

The Multiple Terminal Manager provides the capability to define transactions and manage the programs which support those transactions. It also provides management of multiple terminals as needed to support these transactions and their various application programs. The components of the Multiple Terminal Manager are:

- A program/storage manager, which controls the execution and flow of the application programs within a single program area.
- A terminal/screen manager, which controls the presentation of screens and communications between terminals and application programs.
- A file handling mechanism, which simplifies the storage and retrieval of data on the bulk storage devices.

The terminal manager simplifies such transactions by:

- Automatically providing input and output buffers for the application program.
- Performing I/O operations to access fixed screen formats from the screen file. Here, the term screen refers to the image that is displayed on the screen of an IBM 4979 or IBM 4978 Display Station. Fixed-screen formats consist of unmodifiable text and definitions

of possible areas for data input. Screens are built using the $IMAGE system utility.

- Returning control to the user program to allow modification of the input buffer containing the screen if desired.

- Performing the set of I/O operations involved in writing the screen to the terminal, filling in unprotected fields with user-defined output data, and reading the data entered by the operator before returning control to the application program that requested the action. The terminal manager assumes that each action request involves both output and input operations, thus eliminating the need for the application program to make separate requests.

In addition, the Multiple Terminal Manager provides storage, file, and program management services, terminal transaction statistics, the capability for sign-on programs for password validation, and error recovery for I/O and program check conditions.

Multiple Terminal Manager applications can be written in the Event Driven Language, Series/1 Assembler Language, COBOL or FORTRAN IV. Disk I/O can be performed by an application program using indexed or direct access methods. Terminal support is provided for locally attached IBM 4979 and 4978 Display Stations and remote Teletype[1] ASR 33/35 compatible terminals attached using a single-line or multiline asynchronous communication adapter.

The disk I/O function provides the following for disk and diskette files:

- Indexed Access Method file support

- Direct file support

- Storage conservation through automatic open and close functions.

The TERMINAL file describes the terminals to run with the terminal manager. In this file, you specify the terminal type, the name of the terminal, the screen to be used as the primary menu screen, whether or not sign-on is required, and for asynchronous terminals, the length of the input buffer. This flexibility allows you to add or delete terminals without rebuilding the terminal manager.

---

[1] Trademark of the Teletype Corporation.

Your application programs can be executed by way of a selection from the terminal menu or by a program. Only the program name is required. The Multiple Terminal Manager performs the operations necessary to load the program and control its execution.

Screen formats are used by application programs and the Multiple Terminal Manager itself. Each screen is a data set in a volume that defines protected fields and optionally defaults for unprotected fields. Three screens are predefined:

- The initial program load (IPL) screen that is displayed when the Multiple Terminal Manager task set starts.
- The sign-on screen (displayed if a sign-on procedure is specified for the terminal).
- A sample primary menu screen for program selection. However, you can select any screen as a menu screen.

These screens are provided as samples and can be modified to suit individual requirements. (You can define additional screens using the $IMAGE screen build utility).

The Multiple Terminal Manager responds to an interrupt from a terminal by loading the requested program specified by program name or program function key selection. The terminal manager routes subsequent operator entries to the associated program. Two program function keys are reserved:

- PF3 - signals the Multiple Terminal Manager to terminate the current program and display the menu screen
- PF6 - signal can still be defined to print the contents of the current screen on the system printer.

HARDWARE

The minimum hardware configuration required for the Multiple Terminal Manager is as follows:

- Series/1 processor (either 4952 or 4955) with 96KB storage
- Disk storage device (either 4962 or 4963)
- An Event Driven Executive $SYSLOG device. It is recommended but not mandatory that this be a non-Multiple Terminal Manager terminal for receiving any system

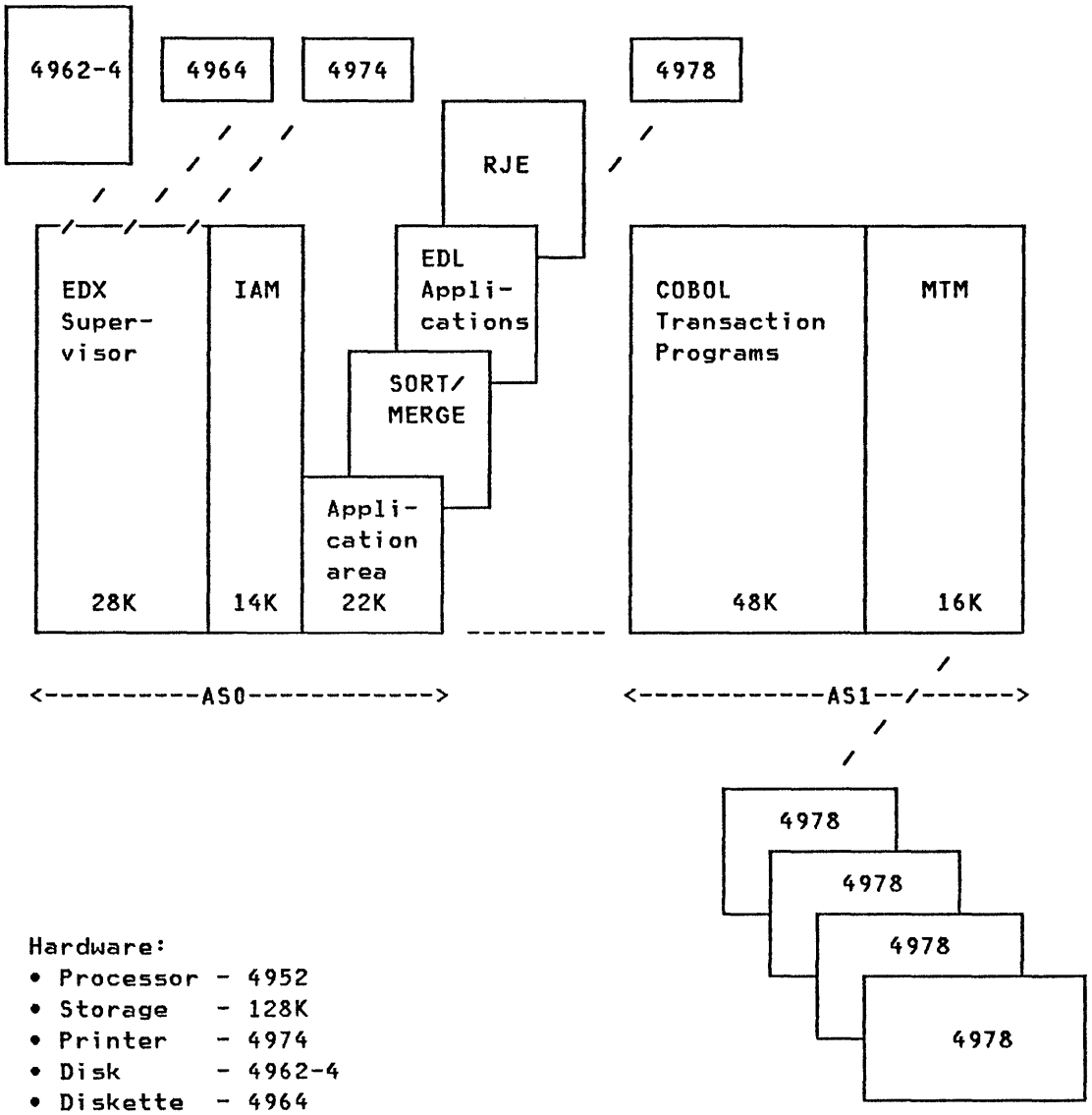
messages during the Multiple Terminal Manager execution.

- Display station (either 4978 or 4979)
- Printer - IBM 4973 or IBM 4974 (to define a $SYSPRTR for error messages).

The following optional hardware is supported:

- Additional 4978 or 4979 terminal devices
- Teletype[2] devices connected to an ACCA adapter
- Printers as supported by EDX Version 1.1
- Additional direct access devices (disk or diskette) as supported by EDX Version 1.1
- Additional storage as supported by EDX Version 1.1.

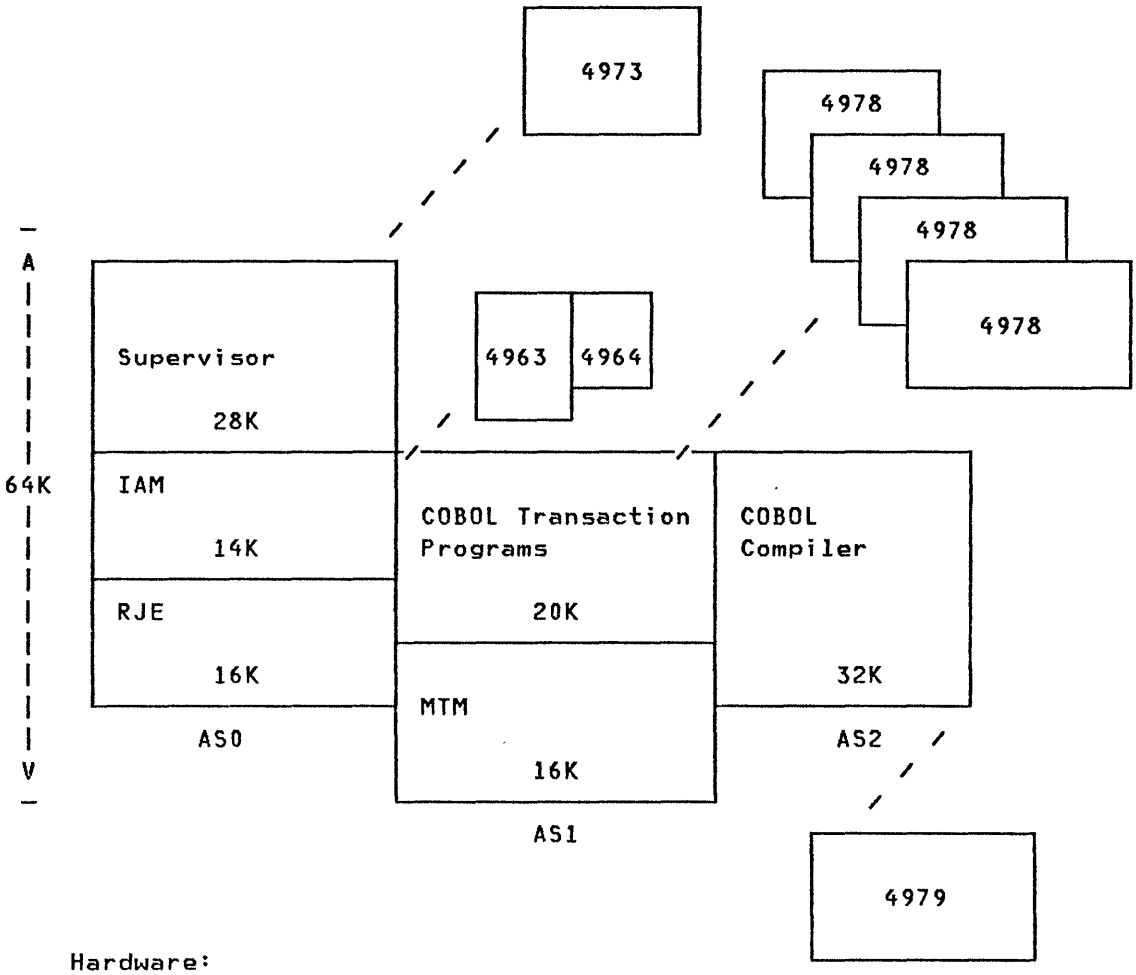
[2] Trademark of the Teletype Corporation.



Hardware:
- Processor - 4952
- Storage - 128K
- Printer - 4974
- Disk - 4962-4
- Diskette - 4964
- Displays (5) - 4978

Software:
- Event Driven Executive V1.1
- Indexed Access Method (IAM)
- COBOL Library
- Multiple Terminal Manager (MTM)
- Remote Job Entry (RJE)
- Event Driven Language (EDL)

Figure 2. Transaction Oriented System



Hardware:

- Processor - 4955
- Storage - 128K
- Printer - 4973
- Diskette - 4964
- Disk - 4963
- BSC single line - #2074
- Displays (5) - 4978

Software:

- Event Driven Executive
- Indexed Access Method (IAM)
- COBOL Compiler
- COBOL Library
- Multiple Terminal Manager (MTM)
- Remote Job Entry (RJE)

Figure 3. Transaction Oriented/Program Development System

SOFTWARE

The minimum software requirements for executing the Multiple Terminal Manager under EDX are:

- EDX Supervisor Version 1.1 (5719-XS1)
- EDX Utilities (5719-UT3).
- Program Preparation Facilities (5719-XX2), required for program preparation and installation of Multiple Terminal Manager applications

Additional software supported by the Multiple Terminal Manager includes:

- EDX Indexed Access Method (5719-AM3) if indexed I/O will be used
- EDX FORTRAN (5719-F02) if FORTRAN applications are to be executed
- EDX COBOL (5719-CB3 and 5719-CB4) if COBOL applications are to be executed.

PROGRAM OPERATION

The Multiple Terminal Manager is invoked using the $L command. This will cause the Multiple Terminal Manager program manager to be loaded into storage and activated. The first program activated by the program manager is the initialization routine.

MULTIPLE TERMINAL MANAGER INITIALIZATION PROGRAM

This program determines the number of terminals that are being controlled and prepares the tables and in-storage control blocks to support those terminals. The initialization routine loads and initializes a terminal server for each terminal that is to be controlled by the Multiple Terminal Manager. When initialization is complete, control is returned to the program manager.

TERMINAL SERVER PROGRAMS

The terminal server programs perform all input/output and interrupt handling functions for those terminal devices operating under the control of the Multiple Terminal Manager. There is one terminal server program for each terminal assigned to the Multiple Terminal Manager.

When a server program is first activated by the initialization program, it will display a base Multiple Terminal Manager message screen and wait for an interrupt from the operator. If a sign-on procedure is required for this terminal, the server will invoke the sign-on procedure and wait until an operator has signed on properly. After proper sign-on, the server will perform I/O operations as needed to satisfy the requirements of operator dialogues and the associated application program. The terminal servers use ENQ/DEQ and WAIT/POST mechanisms for interfacing with the program manager.

APPLICATION PROGRAM MANAGER

The application program manager provides the Multiple Terminal Manager program management facilities required to satisfy operator requests. The program manager controls the contents of the program area and the initialization of programs within the area.

The Multiple Terminal Manager is a transaction processing subsystem which executes as an application program under the Event Driven Executive. The Multiple Terminal Manager transactions are initiated by a terminal operator using a transaction selection menu (also referred to as a program selection menu). Programs can request single or multiple operator prompts, process the input, and then request additional input or terminate.

The Multiple Terminal Manager applications are automatically connected to a terminal when a transaction begins. The Multiple Terminal Manager in turn processes terminal I/O for the these applications. The applications execute within the managed program area, and are provided program, terminal, screen and file management services through calls to the Multiple Terminal Manager. For example, a program executing under control of the Multiple Terminal Manager displays a menu screen offering the operator a choice of functions. Based on the operator's selection, the application program then performs processing operations, such as reading information from a data file, displaying the data at the terminal, and waiting for the next response.

PROGRAM MANAGEMENT

The program management facilities provide the capability for the control of application programs while performing their respective transactional processes within a single program area. Because all of the Multiple Terminal Manager application programs operate in the same program area, the Multiple Terminal Manager program management facilities contain the support needed to allow multiplex operation and sharing of the program area.

The application programs using these program/storage management facilities will always have the following four items associated with them:

1. The application program itself. This is the user-written code that performs the transaction processing as required by the user. It will reside in the programs file and be loaded into the in-storage program area by the manager.

2. The "swap out" data set. This data set is used by the manager to save programs and data across calls to ACTION, LINK, LINKON, CYCLE, and WRITE.

3. The input buffer. This buffer contains the data that was last entered by the operator when the current segment of the application program is entered. This buffer also contains the protected characters of the screen display that the application program is preparing for the next dialogue with the operator. Refer to Figure 4 for additional information. This buffer is automatically allocated by the Multiple Terminal Manager and is 1920 bytes long.

4. The output buffer. This buffer contains the unprotected characters of the screen display that the current application program is preparing for the next dialogue with the operator. These unprotected characters can either be default values or the response to an operator query. Refer to Figure 4 for additional information. This buffer is automatically allocated by the Multiple Terminal Manager and is 1024 bytes long.

The application programs will interface with these facilities using the callable functions described in the following paragraphs.

ACTION - Fetch Operator Response

The ACTION function provides the application program with the ability to display a screen and then to obtain the operator's input to a display on the terminal screen. This call function should be preceded by one of the calls to format the display buffer.

When this function is called, the manager saves the information needed to return to the next sequential program instruction when the ACTION function is complete. The manager then communicates with the operator at the proper terminal and constructs the buffer to be returned to the calling program. While the manager is performing this operation for the calling program, the storage space containing the calling program might be placed (by the manager) with one of the other application programs, thus allowing the sharing of the storage resource among several application programs. When the requested ACTION function has been completed and the storage has been freed by another application program, the manager will reload the calling application program and return control to that program at its next sequential instruction. Figure 4 shows the status of the input and output buffers when this function is called.

LINK - Load and Execute Program

The LINK function enables an application program to complete its own execution by loading and executing some other application program. Once an application performs a CALL LINK, that program will not be returned to from the manager, unless the manager detects an error in the calling sequence to the LINK function; consequently, instructions following a CALL LINK must be included to handle this problem. Refer to Figure 5 for more information. Also, Figure 4 shows the status of the input and output buffers when this function is called.

LINKON

The LINKON function is a combination of the functions provided by the ACTION and LINK functions; that is, it requests an operator action and, when this action is complete, loads and executes some other application program. Figure 4 shows the status of the input and output buffers when this function is called.

Buffer contents upon return from	Input buffer	Output buffer
- Terminal operator	Data entered by operator	Blanks (X'40')
- CALL LINK	Blanks (X'40')	Unchanged from calling program
- CALL LINKON	Data entered by operator	Blanks (X'40')
- CALL CYCLE	Blanks (X'40')	Unchanged
- CALL SETPAN	Protected data from new screen panel	Unprotected data from new screen panel

Buffer contents upon program exit by	Input buffer	Output buffer
- CALL ACTION	Write protected data if CALL SETPAN issued	Scatter written into unprotected fields on screen
- CALL LINKON	Same as by CALL ACTION	Same as by CALL ACTION
- CALL LINK	Write protected if CALL SETPAN issued	Saved
- CALL CYCLE	Same as by CALL LINK	Same as by CALL LINK

Figure 4. Buffer contents and usage during control of the Multiple Terminal Manager program/storage manager.

CYCLE - Suspend Current Terminal Application

The CYCLE function provides an application program with the capability of suspending itself to allow other appli-

cations or terminals to become active. After other applications have had an opportunity to execute, the manager will reload the calling program and return control to it at the next sequential instruction after the call to CYCLE. Refer to Figure 4 for more information.

MENU - Return to Multiple Terminal Manager Control

The MENU function provides the application programs with the capability of aborting their own operation and returning to the control of the Multiple Terminal Manager base program with the operator selection menu displayed on the terminal. The operator at a terminal can perform this same function with the PF3 key on his terminal.

WRITE - Output to an Asynchronous Terminal

The WRITE function is provided for those applications that use asynchronous terminals such as the Teletype[3] ASR 33. It causes the specified buffer contents to be displayed on the specified terminal. After the display has been written on the terminal, the application program must then perform a CALL ACTION to receive responses back from the operator. This function executes in a similar fashion to the functions described under "Storage/Program Management" in that the application program does not remain in storage while the buffer is being written.

Terminal/Screen Management

The terminal screen management facilities provide you with a simplified method of performing the terminal handling functions that your application program may require. In addition to providing the means of operating with the Series/1 keyboard/CRT devices, the terminal/screen manager also provides facilities for handling asynchronous terminal devices. Application programs using keyboard/CRT terminals may interface to the terminal/screen management facilities through four callable functions, while the interfaces with asynchronous terminals will use only one callable function.

---

[3] Trademark of the Teletype Corporation.

SETPAN - Format the Input and Output Buffers

The SETPAN function provides the application program with the ability to request that a specified screen be retrieved from the screens file and loaded into the input buffer. The screen images in the screens file will have been built using the $IMAGE utility. The contents of this input buffer will then be displayed on the next call to ACTION, LINK, or LINKON. The screen will be written on the terminal display with all positions write-protected except those in which nulls (X'00') were defined.

When this function is called, one of the parameters will specify the name of the screen to be retrieved.

SETCUR - Move Cursor to Specified Position

The SETCUR function provides the application program with the ability to identify the character position at which the terminal/screen manager will display the cursor when the screen is displayed.

BEEP - Set Audible Alarm

The BEEP function provides the application program with the ability of activating the audible alarm on the next CALL ACTION as a signal to the terminal operator. This function is ignored for those terminals that do not have an audible alarm capability.

CHGPAN

The CHGPAN function is used to notify the terminal manager of changes to the number of protected/unprotected characters of a screen panel in the input buffer so that the terminal manager will know how many unprotected data characters to write on the next output cycle.

Figure 5 is an example of an application program that performs MTM functions to converse with an operator, to allow other applications to run, and to link to another application program.

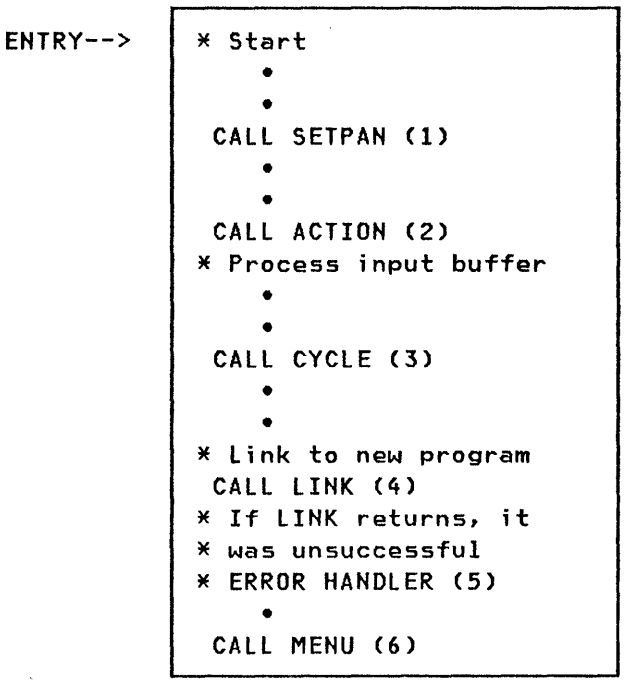


Figure 5. Sample Application Program

1. Read screen image from data set. Protected data goes into the input buffer and unprotected data goes into the output buffer.

2. The ACTION function writes protected data from the input buffer, and writes unprotected data from the output buffer. It then reads operator inputs into the input buffer.

3. The CALL CYCLE suspends the program to allow others to execute.

4. The last call in this application program terminates the program and starts the next one.

5. An error handling routine in the event that the manager detects an error in the calling sequence for the LINK function.

6. The CALL MENU within the error handling routine terminates the current program and returns to the Multiple Terminal Manager with the basic menu displayed.

File Management

The file management facilities provide common, easy-to-use support for all disk data transfer operations as needed for the application programs. These facilities provide support for both indexed and direct files under the control of a single callable function FILEIO (Perform Disk I/O)

The FILEIO function performs read and write operations on the disk using either indexed or direct accessing of the information. You specify the desired operation to the program using a file control area. This is a callable function, and the calling program will remain in storage while the operation is taking place.

FILEIO

FILEIO provides the facility to access previously created files using the call interface described earlier. These files must have been previously defined and loaded using an offline user-written utility.

```
CALL FILEIO,(FCA),(BUFFER),(RETURN CODE)
```

Figure 6. FILEIO

The calling parameters are:

- File Control Area (FCA) - Address of a table with the parameters describing the requested operations:
- BUFFER - Address of the user program I/O buffer. This is in the user program space. FILEIO and Indexed Access Method have their own buffers.
- RETURN CODE - Address of the 2-byte field to contain the return code passed back by FILEIO. This can be a FILEIO return code, or it can be passed through from the Indexed Access Method.

Request type
Data set name
Key relation operator or number of records
Key length
Key location or logical EOD record number
Relative record number
Volume name

Figure 7. File Control Area

Indexed File Request Types

Type	Function
RELS	Release from sequential processing mode
RELR	Release from random processing mode
PUTU	Put operation, update mode
PUTD	Put operation, delete mode
PUTN	Put operation, new mode-adds a record to the file
GETD	Get operation, direct read
GETS	Get operation, sequential read
IDEL	Delete operation
ICLS	Close an indexed data set
GTRU	Direct get, update mode
GTSU	Sequential get, update mode

Direct File Request Types

Type	Function
READ	Read the record defined by the RRN field of the FCA into the user-provided buffer.
WRIT	Write the record defined by the RRN field of the FCA into the major user-provided buffer.
SEOD	Set the system-maintained EOD pointer to the record number provided in the relative record number field of the FCA.

Multiple Terminal Manager Utilities

The utility program support consists of operator service functions that assist you in the operation of the Multiple Terminal Manager system.

Terminal Activity Report. This report utility enables you to display the names and current status of the terminals under control of the system.

Terminal Connection Facilities. Provides the operator with the facilities to disconnect and reconnect terminals during the normal Multiple Terminal Manager operation. These services are performed by the two operator commands:

- Disconnect - Turn Off Specified Terminals. This facility provides the operator with a means of shutting down individually-specified, or all terminals, on the Multiple Terminal Manager system. If the operator requests that a terminal that is currently involved in a transaction is to be disconnected, that terminal will be allowed to complete its associated transaction before being disconnected.
- Reconnect - Turn On Specified Terminals. This facility provides the operator with a means of restoring a disconnected terminal back into operation.

Programs Report. Prints information about available programs.

PROGRAMMING CONSIDERATIONS

The Multiple Terminal Manager applications are processed as initial tasks of a program which execute within the program manager's overlay area. On the first execution of a program during a transaction, the program is brought into the overlay area by a LOAD. Then, when the program returns to the Multiple Terminal Manager through a CALL ACTION, WRITE, CYCLE, MENU, LINK or LINKON, the Multiple Terminal Manager dequeues the program from the system using DETACH. Also, if the program returned using a CALL ACTION, WRITE or CYCLE, the Multiple Terminal Manager writes the program out to the MTMSTORE data set. The overlay area is then free for use by other programs. When the Multiple Terminal Manager is ready to reexecute that program for subsequent processing of the transaction, the program manager reads the program into the overlay area and requeues that program to the system using ATTACH.

Thus, the Multiple Terminal Manager transaction programs should adhere to the following conventions:

- No subtasks should be active across calls to the Multiple Terminal Manager
- No system-wide resources should be enqueued across calls to the Multiple Terminal Manager
- Transaction programs cannot use overlays
- Transaction programs should be written as main programs with the expectation of receiving four parameters at initiation
- Transaction programs should use the Multiple Terminal Manager for all terminal and disk I/O
- All other I/O should be complete across calls to the Multiple Terminal Manager
- Transaction programs should terminate only with calls to the Multiple Terminal Manager and should not issue any PROGSTOP, ENDTASK, or DETACH instructions
- Error handling routines should terminate with a CALL MENU.

MTMSTORE

MTMSTORE, the Multiple Terminal Manager work file, contains:

- The Multiple Terminal Manager program table
- The Multiple Terminal Manager screen table
- A program and buffer save area for each terminal defined in the TERMINALS specifications file.

The size of the MTMSTORE file can be calculated as follows:

- Allow 10 bytes per screen in the SCRNS volume; round up to the nearest sector
- Allow 12 bytes per program in the PRGRMS volume; round up to the nearest sector
- Allow per terminal:
  1. Enough sectors to hold a copy of the largest program in the PRGRMS volume
  2. Four additional sectors for full screen devices.

This data set is in the volume MTMSTR.

TERMINAL

This file is built with the $FSEDIT system utility. It contains one record/terminal containing the specification of a terminal.

Dvtp,Termname,Menuscrn,Y/N

Dvtp Type of terminal. Enter:

```
4979 IBM 4979 full screen
4978 IBM 4978 full screen
3335 ASR 33/35 line at a time
```

Termname 1- to 8-character name of a terminal. This name must be identical with the device name specified on the EDX TERMINAL statement at SYSGEN. This name should not be the $SYSLOG device.

Menuscrn The data set name of the screen to be displayed after an operator exits a transaction or signs on. For asynchronous terminals, this field is

ignored.

Y/N — Y = This terminal is required to use the SIGN-ON and SIGNOFF programs. If a user program named SIGN-ON does not appear in the program library, this terminal is not usable. N = This terminal is always signed on.

Comment records are acceptable in this file as well as comments following specification records. Comment records must have an * in position 1.

An example of this file would be:

```
4979,DISPLAY1,MENUSCRN,n
4978,DIS49780,MENUSCRN,y
3335,ACCA1,MENUSCRN,y
/*
```

End of specifications must be indicated with a /* record.

Before processing each record during task set startup, the record is listed on $SYSPRTR. When startup is complete, all terminals will have the Multiple Terminal Manager IPL screen displayed. The terminal data set is in the volume PRGRMS.

Screen Format Volume - SCRNS

This volume contains screen panel data sets for full screen images built using the $IMAGE system utility. These screens must have been built with a 24x80 dimension size.

User Application Program Volume - PRGRMS

All programs loaded by the Multiple Terminal Manager are loaded using the names of the data sets on this volume. The terminal file is also in this volume.

Application programs are stored in this volume as the output of the $UPDATE utility. The names of the programs are the names used by the operator from the MENU mode to invoke programs and can also be used as the program parameter on a CALL LINK from one program segment to another.

When the Multiple Terminal Manager is initiated, a program table is built containing the name of each program data set in the PRGRMS volume.

Each program is checked to determine if it is too big for the program area in the Multiple Terminal Manager. If this occurs, the user should split the program into segments using LINK or increase the size of the program area.

Sign-On File - SIGNONFL

This file contains sign-on records for use by the SIGN-ON program. The format of the file is:

Field Name	Positions	Contents
EMPLOYEE	1-08	Employee number
PASSWORD	9-12	Password
USERID	13-16	User ID
USER CLASS	17-20	User class
NAME	21-32	Employee name

This file is built by using the $FSEDIT EDX utility. This file is in the volume PRGRMS. A /* in columns 1 and 2 denote the end of the file.

OPERATOR INTERFACE

Multiple Terminal Manager Initialization

The Multiple Terminal Manager can be initiated from any terminal defined to the system by entering the $L $MTM,PRGRMS command. The $L $MTM command will cause the Multiple Terminal Manager program manager to be initiated. The program manager will then initiate a terminal server for each terminal specified in the TERMINAL file. Upon completion of initiation, the IPL screen, IPLSCRN, will be displayed at each of the Multiple Terminal Manager terminals. IPLSCRN will specify that the operator press the enter key in order to display either the sign-on or menu screen.

The data set IPLSCRN is displayed on each full screen terminal after the Multiple Terminal Manager is started. It requests that the operator press the enter key to connect the terminal to the Multiple Terminal Manager. It should not be displayed again.

For execution	VOL ID	Data set name	Size
Swap data set	MTMSTR	MTMSTORE	See MTM data set reqm'ts
Program manager	PRGRMS	CDMPGMGR	36 Sectors
4978/4979 term server	PRGRMS	CDMFLSCR	6 Sectors
TTY term server	PRGRMS	CDMTTY	6 Sectors
MTM initialization	PRGRMS	CDMINIT	22 Sectors
Terminal specifications	PRGRMS	TERMINAL	4 Sectors (appx)
User application pgms	PRGRMS	(User specified)	?
		?	?
		.	.
		.	.
		.	.
Screen formats	SCRNS	(User specified)	4 Sectors
		SCREENS	Per screen
Sign-on file	PRGRMS	SIGNONFL	4 Sectors (appx)
For program preparation			
MTM stub	(User specified)		8 Sectors
MTM auto call list	(User specified)		2 Sectors
For rebuilding MTM			
MTM Source	(User specified)		2500 Sectors

Note: MTMSTR,PRGRMS,SCRNS must be defined at SYSGEN time as logical volumes.
This may necessitate a remapping of the disk on which they will reside.

Figure 8. Summary of Multiple Terminal Manager Data Set Requirements.

SIGN-ON

If sign-on was specified for the terminal, then the sign-on screen, sign-on, will be displayed following the IPL screen. The sign-on screen will require that the operator enter a user-id and password. After sign-on processing has been completed, then the menu screen will be displayed.

PROGRAM INITIATION AND TERMINATION

After the Multiple Terminal Manager initiation and sign-on processing have been completed, then the menu screen is displayed. The menu screen is the screen from

which the operator can initiate transactions. A transaction is initiated by the operator entering either a program name or pressing a PF key when the menu screen is displayed. A PF key will initiate program PF0n where n reflects the number of the PF key pressed. If data is entered, the Multiple Terminal Manager will consider the first eight bytes to be a program name.

After a transaction is initiated, the operator can terminate it by pressing the PF3 key. Upon termination of the transaction, the menu screen will be redisplayed. Subsequently pressing the PF3 key from the menu screen will cause the sign-on screen to be redisplayed if sign-on was specified for that terminal. Otherwise, PF3 will be a no-op and the menu screen will remain displayed.

DISCONNECT

Terminals can be disconnected from the Multiple Terminal Manager or the Multiple Terminal Manager can be terminated via the DISCONNECT facility. DISCONNECT is invoked from the menu screen by keying in either DISCONNECT,termname or DISCONNECT,ALL. If a referenced terminal is in a transaction, that transaction is allowed to complete. When the terminal returns to MENU state, it is automatically signed off and displays the YOU ARE DISCONNECTED message. Terminals in MENU state are signed off immediately.

If ALL is specified, all terminals are disconnected. When the last terminal is truly disconnected, whether using ALL or separate disconnects, the manager task is stopped.

To enter this command from a menu screen, the screen must contain at least nineteen unprotected characters.

While a terminal continues in a transaction with disconnect pending, the audible alarm is sounded after every interaction to tell the operator that a disconnect is pending.

RECONNECT

If the referenced terminal is disconnected, it is reconnected in a signed-off status if applicable. If it is not disconnected, the command is ignored.

PROGRAMS REPORT

This report displays data from the program table about each available program. It is intended mainly for debugging during development of the manager but is included as a working example for possible use.

The program name for this program is PGMRPT.

TERMINAL ACTIVITY REPORT

This program displays the names and status of all terminals on the system. If more than 22 terminals are attached, the operator must press ENTER to page to successive groups of 22 lines.

The program name for this program is REPORT.

SCREEN PRINT

Terminal displays can be printed on the system printer using the PF6 key to invoke the system's print screen facility. EDX print screen facility. This entails pressing the PF6 key.

DISTRIBUTION AND INSTALLATION

The Multiple Terminal Manager will be distributed as a licensed program and will consist of the following items:

- Prebuilt Multiple Terminal Manager - This will be a ready to load program consisting of a program manager, file manager, terminal servers and sample programs.
- Multiple Terminal Manager source - This will be the complete set of the Multiple Terminal Manager source code for the user who wants to tailor his Multiple Terminal Manager environment.
- Application Stub - This will be the Multiple Terminal Manager stub in object format that the user must include with his application programs at link time.

- Application Stub AUTOCALL List - This is the auto list that the user will use at link time to cause the application stub to be appended to his program.

- Screen Formats - This will be a set of screens to support the default Multiple Terminal Manager and sample programs.

- Terminal File - This will be a set of miscellaneous terminal statements to support the default system.

The user must create the following volumes on his system disk:

- PRGRMS - This volume is for the Multiple Terminal Manager programs, user application programs and the terminal specifications file.

- SCRNS - This volume is for the screen formats used by the Multiple Terminal Manager and user applications.

- MTMSTR - Contains only the MTMSTORE data set. This the Multiple Terminal Manager swap file.

After the volumes have been created, the user can then copy the prebuilt Multiple Terminal Manager, screen formats and terminal file from the shipped diskettes to disk. This will install the default Multiple Terminal Manager and establish the following data sets.

Data sets within the PRGRMS Volume:

CDMPGMGR Multiple Terminal Manager program manager

CDMSVR89 Multiple Terminal Manager full screen, 4978 and 4979, terminal server

CDMSVR33 Multiple Terminal Manager TTY terminal server

CDMINIT Multiple Terminal Manager initialization routine

TERMINAL Multiple Terminal Manager terminal specification file

Other Miscellaneous data sets needed for the sample programs

Data sets within the SCRNS Volume:

IPLSCRN The initial Multiple Terminal Manager started screen

SIGN-ON The sign-on screen

MENUSCRN The default menu screen

Other Miscellaneous screen data sets needed for the sample programs

If the user wants to tailor his Multiple Terminal Manager, then he can re-assemble, re-build and replace the changed Multiple Terminal Manager components.

The user can then modify his terminal specifications file to match his system environment by using the $FSEDIT system utility. He can also add screen formats to the SCRNS volume using the $IMAGE system utility.

Before executing the Multiple Terminal Manager, the user has to create the MTMSTORE data set; refer to Figure 8 for the MTMSTORE requirements.

PROGRAM PREPARATION

During program preparation, the user must ensure that the link phase includes the Multiple Terminal Manager shipped AUTOCALL list and application stub in order to resolve CALLs to the Multiple Terminal Manager function routines.

After linking his application programs with the Multiple Terminal Manager application stub, the user then must store the prepared program in the PRGRMS volume using $UPDATE.

PERFORMANCE INFORMATION

The information shown in Figure 9 may be used for performance planning purposes. The basis for the information is from modelling the Multiple Terminal Manager system. While the modelling results provide some general guidelines as to the performance of the system, it should be noted that the models are only partially validated for the workloads modelled. Other workload assumptions may provide a different set of performance results.

As a general guideline to Multiple Terminal Manager performance, the mean expected response time should be less than or equal to 3 seconds with 1 to 4 terminals. The performance may exceed this range. Mean expected response time is the average response time experienced from when the user presses the enter key until the first character of the response is displayed.

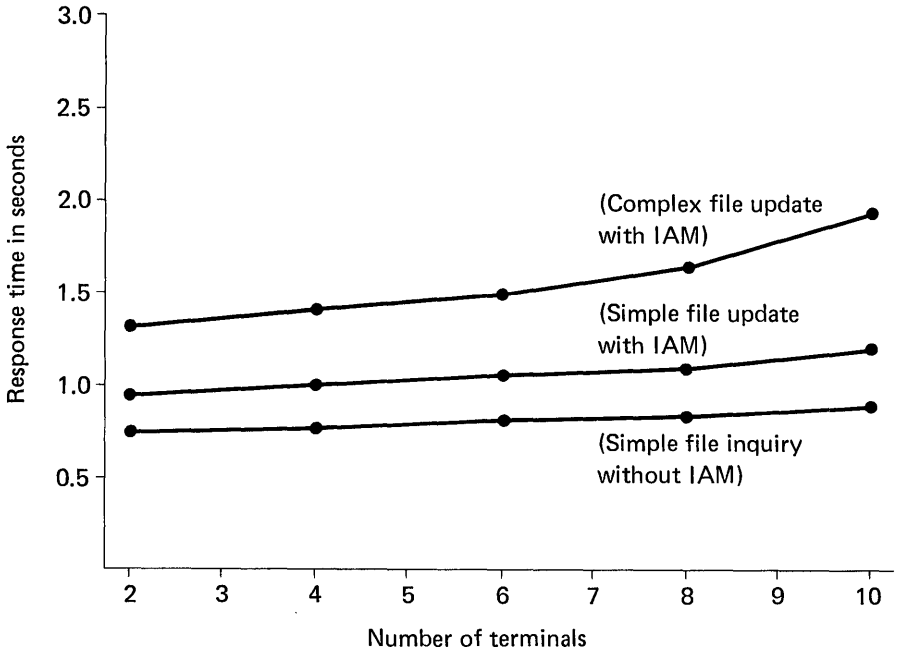


Figure 9. Multiple Terminal Manager Performance Trends

Figure 9 shows the response time results for each of the transactions. The highest set of response times was achieved under a complex file update transaction and the lowest was with a non-Indexed Access Method simple file inquiry. With all these cases the response time was under 2 seconds.

A key variable to response time results is the think plus key-in-time associated with a set of terminals. When this variable is very short, the system resources may be stressed and can result in higher response times. When this variable is increased, the mean response time will be smaller.

Refer to Figure 10 for a definition of the transactions used in the model. The think mean plus key-in-time for the modelling was 20 seconds.

Type	Program size	IAM Database	Number of disk I/O	Number of application instructions executed
Simple inquiry	22K	GETD	4	27.8K
Simple file inquiry/ update	22K	GETD, PUTU	5	28.7K
Simple file inquiry (non-IAM)	15.6K	-	2	15.6K
Complex file update	48K	GETD, GETD, PUTU	10	41.9K

Figure 10. Transaction Description

The following were the modelling assumptions:

- Configuration

  -4955 (translated)
  -4963 (1 unit)
  -4978 display terminals
  -Adequate storage
  -Indexed Access Method

- Workload

  -Transaction rate of 3 per minute per terminal
  -20 seconds (exponential) think plus key-in-time
  -COBOL and Indexed Access Method application

OVERVIEW

The Indexed Access Method Licensed Program (5719-AM3) is a data management system that operates under the IBM Series/1 Event Driven Executive. It provides you with an Event Driven Language callable interface to build and maintain an indexed data set and to access, by key or sequentially, your records in that data set. In an indexed data set, each of your records is identified by the contents of a predefined field called a key. The Indexed Access Method builds into the data set an index of keys that provides fast access to your records.

INDEXED DATA SET FEATURES

For your applications using indexed data sets, the Indexed Access Method offers several useful features:

- Direct and sequential processing. Multiple levels of indexing are used for direct access, and sequence chaining of data blocks is used for sequential access.
- Support for high insert and delete activity without significant performance degradation. Free space is distributed throughout the data set and in a free pool at the end so that inserts can be made in place; space provided by deletes can be immediately reclaimed.
- Access to a single data set by several requests concurrently. These requests can execute from the same or different programs. Data integrity is maintained by a file-, block-, and record-level locking system that prevents access to that portion of the file that is being modified.
- Implementation as a separate task. A single copy of the Indexed Access Method executes and coordinates all requests. The buffer pool supports all requests and optimizes the space required for physical I/O; in the user program the only buffer required is the one for the record currently being processed.
- An Indexed Access Method utility program ($IAMUT1) which executes as a user program, and allows you to create, format, load, unload, and reorganize an indexed data set.

DATA PROTECTION

The Indexed Access Method performs all input/output operations by using system functions. Therefore, all data protection facilities offered by the system also apply to the indexed data sets. The following additional data protection is provided by the Indexed Access Method:

- Exclusivity option. As a user of the Indexed Access Method, you specify on the PROCESS or LOAD request that the data set is for use exclusively. This allows you to impose additional control where needed.
- Record locking. The Indexed Access Method automatically prevents two users from using the same data record at the same time.
- Immediate write-back. This optional feature provides the capability of having all updated records written immediately to the data set.
- Accidental key modification. If you attempt to modify the key field in one of your data records, the Indexed Access Method prevents the modification from occurring. This helps ensure that your index and data match.

DEVICES SUPPORTED

The Indexed Access Method supports indexed data sets on these direct-access devices:

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

COMPONENTS

The Indexed Access Method consists of the following components:

- A load module that supports the execution of your programs, which contain the Indexed Access Method requests. Indexed Access Method functions are initi-

ated by your programs through a CALL interface.

- A set of object modules that you may use to generate a customized Indexed Access Method load module (a read only system, for example). If you use the supplied load module, you do not need the object modules.

- An object module, called a link module, which you include with your program to provide the interface to the Indexed Access Method. This link module is sometimes called a stub.

- A set of copy code modules. You can reference these modules from your programs to define symbolic labels used in Indexed Access Method requests.

- A load module for the Indexed Access Method utility program.

FUNCTIONS

You request the services of the Indexed Access Method through a call interface.

(CALL IAM,+FUNC,IACB,(PARM3),(PARM4),+PARM5).

The following functions can be invoked:

PROCESS Builds an indexed access control block (IACB) and connects it to an indexed data set for reading, updating, inserting, and deleting records. You can then use the IACB to issue requests to that data set. A program can issue multiple PROCESS functions and obtain multiple IACBs for the same data set so that the data set can be accessed by several requests concurrently.

LOAD Similar to PROCESS but allows loading or extending the initial collection of records.

GET Directly retrieves a single record from the data set. If you specify the update mode, the record is locked (made unavailable to other requests) and held for possible modification or deletion.

GETSEQ Sequentially retrieves a single record from the data set. If you specify the update mode, the record is locked (made unavailable to other requests) and held for possible modification or deletion.

PUT Loads or inserts a new record depending on how the data set was connected: LOAD or PROCESS.

PUTUP Replaces a record that is being held for update.

PUTDEL Deletes a record that is being held for update.

RELEASE Releases a record that is being held for update.

DELETE Deletes a single record, identified by its key, from the data set.

ENDSEQ Terminates sequential processing.

EXTRACT Provides information about the file (file control block).

DISCONN Disconnects an IACB from an indexed data set, thereby releasing any locks held by that IACB, writing out all buffers associated with the data set, and releasing the storage used by the IACB.

These functions provide you the support necessary to build an indexed data set and to perform direct or sequential processing on that data set. Routines using these services are written in Event Driven Language and can be included in programs written in any language that supports the calling of Event Driven Language routines.

In addition, the Indexed Access Method utility program ($IAMUT1) provides a set of commands which help you manage your indexed data sets. The following commands are provided by the Indexed Access Method utility program:

CR The create function allows you to allocate space for your data set in a volume by internally invoking the $DISKUT1 system utility. When the create command is entered on a terminal, the $DISKUT1 program is loaded and you can then use the AL command of $DISKUT1 to allocate a data set; any other $DISKUT1 function can also be performed. Communication to the $DISKUT1 utility continues until the end command (EN) is entered, at which time communication to the Indexed Access Method utility program is restored. Information on the $DISKUT1 utility can be found in the Utilities, Operator Commands and Program Preparation manual.

DF The define command of the Indexed Access Method utility program uses an existing data set and user-specified information to define an indexed file. When the define command is entered, you are prompted for the immediate write-back option and for the volume and data set names of the data

set to be formatted. The define function performs size calculations and formats the data set. The size calculation information is returned to your terminal at the completion of the define function. Prior to entering the define command, you must use the SE command to set up parameters that determine the size and format of the indexed data set.

DI The display command allows you to display the current saved values for the define command, as described above.

EC The echo command allows you to set or reset echo mode. When echo mode is on, the input and output of the current utility session is logged on the $SYSPRTR device. Echo mode remains on until either the current utility session is ended or it is reset by the EC command. When the Indexed Access Method utility is loaded, echo mode is initially off. Note: Output from the $DISKUT1 program will not be logged since the terminal input/output is handled directly by the $DISKUT1 utility (not $IAMUT1).

EN The end command ends the current utility session.

LO The load command loads an indexed data set from a sequential data set.

RO The reorganize command loads an empty indexed data set from an existing indexed data set.

SE The setparms command prompts you for parameters that determine the structure and size of the indexed data set, and sets these values in a parameter list. Size calculations are performed using the parameters you specify and are returned to your terminal. The setparms command should be entered prior to using the define command to do the actual data set formatting. All values that you specify for parameters of the setparms command are saved (until the end of the session) for later invocations of the setparms command and the define command. You can display these values by entering the DI command.

UN The unload command unloads an indexed data set to a sequential data set.

For each of these utility functions, you are prompted for the volume and data set names of the input and/or output data sets.

You can organize a collection of data into an indexed data set if the data consists of fixed-length records and each record is uniquely identified by the contents of a single predefined field. This identifier is called the key. Records are arranged in an indexed data set in ascending order by key. Some reserved space can be distributed throughout the data set so that records can be inserted in their key position during processing. This space is called free space.

The Indexed Access Method puts records into an indexed data set in either of two ways:

1. In LOAD mode, records are presented in ascending order by key and are loaded into the data set sequentially, skipping any free space. These records are called the base records. Each record loaded must have a new high key; that is, it must have a key higher than any key already in the data set.

2. In PROCESS mode, records are inserted in their proper key position relative to records already in the data set. Records can be inserted using the free space skipped during loading or, if a record has a new high key, in the unused space after the last loaded record.

You connect an indexed data set to your application program with the LOAD or PROCESS request.

The total number of base records that can be loaded is set using the define command of the Indexed Access Method utility program when the indexed data set is built. It is not necessary, however, to load all the base records before processing can begin. The data set can be connected for loading some of the base records, connected for processing including inserts, and later connected for loading more base records. Figure 11 illustrates this sequence.

The total amount of free space for inserts is specified by the define command of the Indexed Access Method utility program when the indexed data set is built. This free space is distributed throughout the data set in the form of free records in each data block, free blocks in each block grouping, and/or in a free pool at the end of the data set.

An indexed data set contains space allocated for base records and insertions and for a multilevel index and the control information required to use the index and the free space. This type of information is useful for planning, diagnostics, and a general understanding of the Indexed Access Method. See "Data Set Format" on page 70 for a description of the format of the indexed data set.

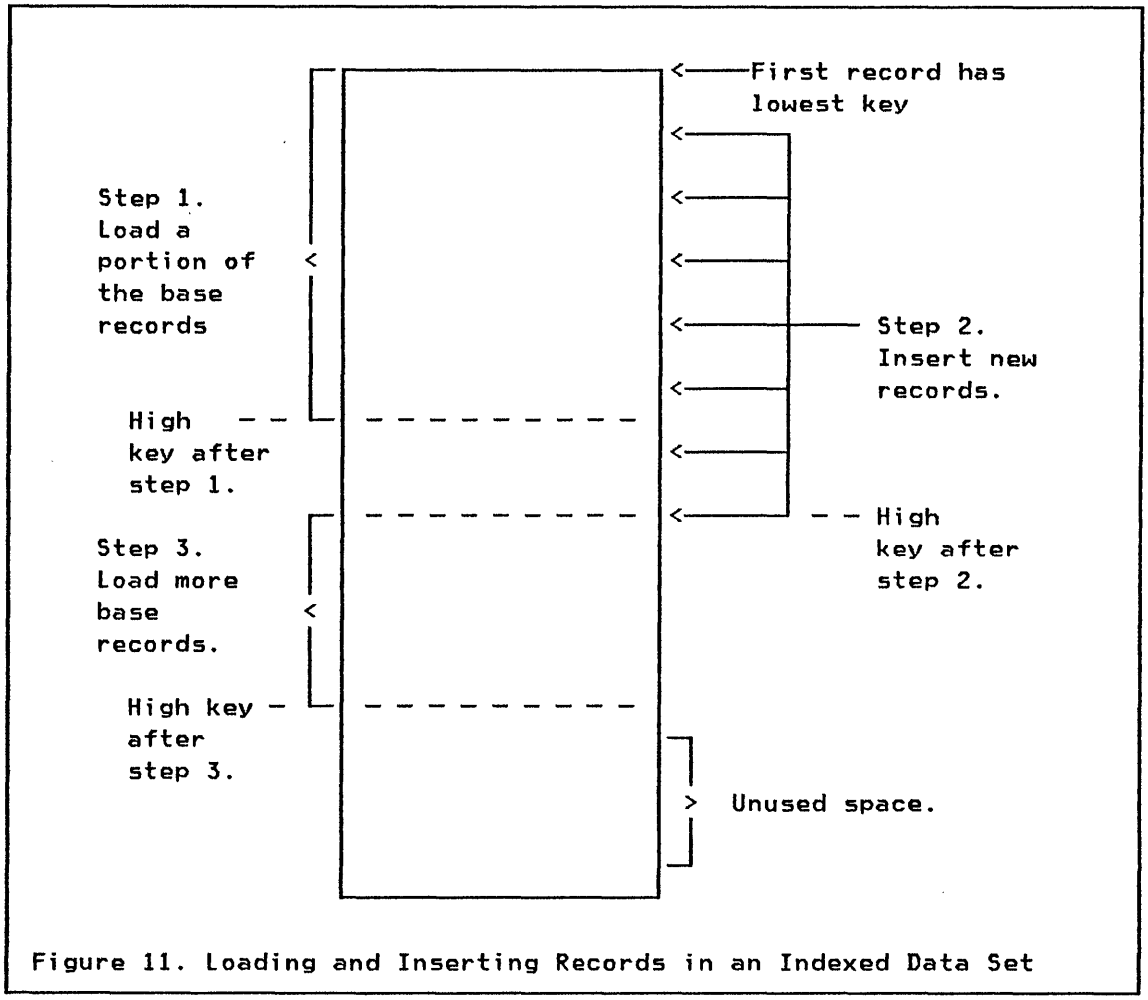


Figure 11. Loading and Inserting Records in an Indexed Data Set

PROGRAM OPERATION

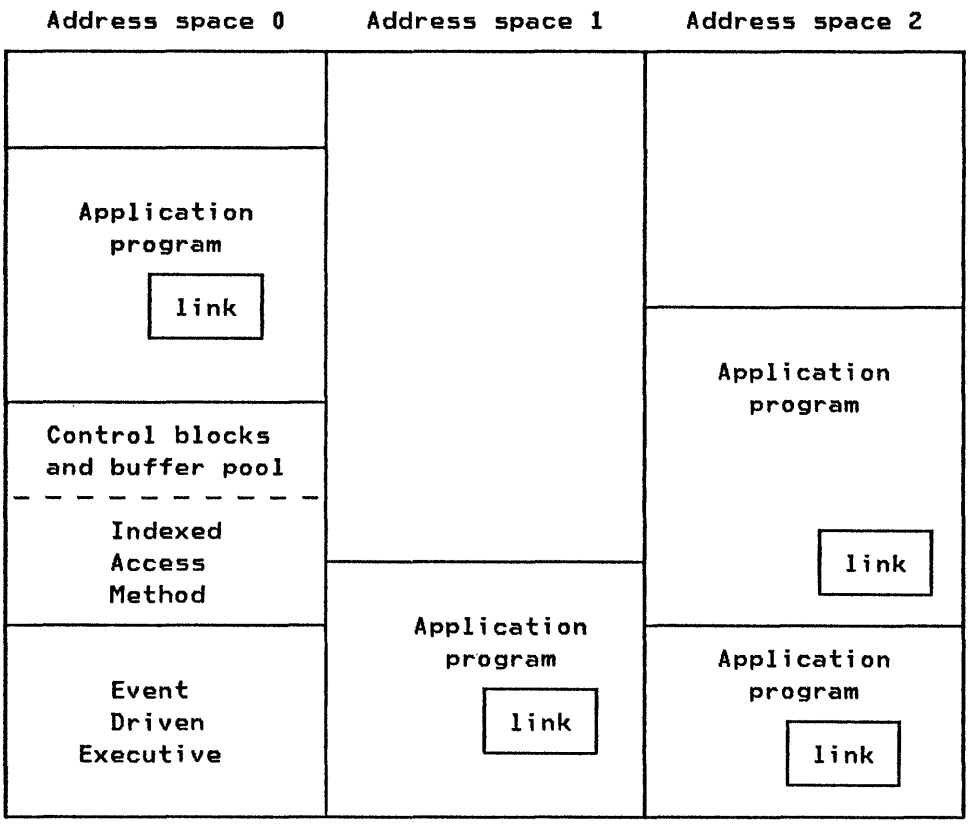
The Indexed Access Method comprises of a load module and a link module. The link module is included in a user's program at link-edit time as an autocall module. The Indexed Access Method performs I/O operations using standard data management requests.

The Indexed Access Method operates under Version 1.1 of the Event Driven Executive.

A single copy of the Indexed Access Method serves the entire system. The Indexed Access Method can be loaded automatically at IPL time through the $INITIAL feature of the system, or it can be loaded manually by using the $L (load) command. The Indexed Access Method does not need to

be loaded before it is used by any program. The name of the Indexed Access Method for loading purposes is $IAM. Once loaded, the Indexed Access Method remains in storage until cancelled.

The Indexed Access Method can be loaded into any address space, including address space zero. It can be invoked (through the link module) from any address space, including the address space it is in. Figure 12 shows an example of a system containing the Indexed Access Method.



Each application program contains a copy of the link module, which provides the interface to the single copy of the Indexed Access Method.

Figure 12. Example System Environment

APPLICATION PROGRAM PREPARATION

Application programs that issue Indexed Access Method requests are prepared as follows:

1. Programs are assembled by any of these assemblers:

   - The EDX assembler, $EDXASM, from the EDX Program Preparation Facility (5798-XX2).
   - The EDX macro assembler $S1ASM (5719-AMA)
   - The Series/1 macro assembler supplied by the System/370 Program Preparation Facility for the Series/1 (5798-NNQ).

2. Use the linkage editor, $LINK, to combine object modules produced by any of the above assemblers, along with the IBM-supplied link object module, into a single module.

3. Use the conversion program, $UPDATE or $UPDATEH, to convert your module into loadable form.

APPLICATION DESIGN INFORMATION

This chapter provides guidelines for designing applications that use the Indexed Access Method. It describes:

- Preparing and maintaining data
- Designing an indexed data set
- Using the functions provided

PREPARING THE DATA

The following sections describe how you can design an indexed data set that uses space efficiently and provides optimum performance.

Defining the Key

Define a single key field for each indexed data set by specifying its size and position in the record when the data set is built by the define command of the Indexed Access Method utility program. The longer the key, the larger the index contained in the indexed data set. Therefore the key should not be longer than necessary. However, the key must be long enough to ensure uniqueness.

Ensuring Uniqueness of the Key. In order to identify each record in an indexed data set, each key must be unique. If key duplication is possible, the key field should be expanded.

Customer name is a good example of a key with which duplications can occur. One way to avoid duplication is to lengthen the key field to include other characters such as part of the customer address or the account number. Since the characters in the key must be contiguous, this solution can involve rearranging the fields in the record.

Another way to eliminate duplication is to modify new records dynamically whenever a duplication occurs during loading or processing. A position at the end of the key field can be reserved for a suffix code. Whenever a duplicate occurs, you can add a value to the suffix and make another attempt to add the record to the data set. The result can be a data set that contains a sequence of keys such as Smith, Smith 1, Smith 2, and so forth. If you add a suffix, you must use the entire unique key to access a specific record.

Providing Access by More Than One Key. To provide good performance with both direct and sequential access, each indexed data set is indexed by a single 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.

One way of providing access by a secondary key is to build a second indexed data set composed of short records that contain only the secondary and primary keys. Using the secondary key to access this data set, the associated primary key can be determined. The primary key can then be used to locate the desired record in the first data set.

Where there are multiple keys to a data set, ensure high performance by selecting as primary key the one that is used most often or the one with which you plan to do sequential processing.

Selecting the Block Size

Records can be blocked in an indexed data set. The block size must be a multiple of 256. Blocking reduces I/O activity; it also allows for free records to be interspersed among base records to provide space for inserts. Free records are one kind of free space; free blocks included at the end of each block grouping, or at the end of the data set, are others.

Specify record size and block size when building the data set by the setparms command of the Indexed Access Method utility program. Each block has a 16-byte header. Therefore, the number of records per block is:

$$\frac{\text{(block size - 16)}}{\text{record size}}$$

The result is truncated; that is, any remainder is dropped. A remainder represents the number of unused bytes in the block. Selection of a block size is largely dependent on record size, but the block size must be a multiple of 256. Other factors to consider are insert activity and buffer space.

Insert Activity. Each block contains allocated record areas into which base records can be loaded and can contain free record areas into which records can be inserted during processing. The ratio of allocated records to free records in a block should be close to the ratio of estimated base records to estimated inserts in the data set. Ideally, block size should be large enough to accommodate enough records to approximate this ratio.

Buffer Space. A large block size is advantageous in that it minimizes the read/write activity, but it is costly in terms of the read/write buffer space required from the buffer pool. Some processing requires a buffer large enough for two blocks.

Examples. A data set consists of 1000 base records with an estimate of 500 records to be inserted and a record size of 70 bytes. Select a block size and a number of free records per block to build an indexed data set.

1. Selecting a block size of 256 with 1 free record per block implies (256-16)/70 = 3 records per block, with a remainder of 30 bytes. The ratio of 2 allocated records and 1 free record accurately reflects the insert activity. Buffer size is minimized. Some space is wasted on the disk (30 bytes per sector). Designing 80-byte records and 256-byte blocks for this data set effectively uses these 30 bytes.

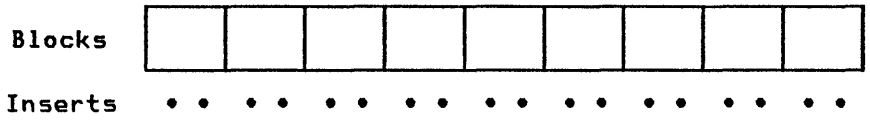
2. Selecting a block size of 512 with 2 free records per block implies (512-16)/70 = 7 records per block, with a remainder of 6 bytes. The ratio of 5 allocated records to 2 free records underestimates the insert activity. The larger block size requires a larger buffer but increases I/O efficiency. Fewer bytes are wasted on the disk (6 bytes in 2 sectors).

Specify free space for inserts using the setparms command of the Indexed Access Method utility program.

An exact calculation to estimate free space is not necessary. Experience can be your best guide; if the need for file reorganization is signalled (no space for an insert) before a major portion of the free space is utilized, you know you must adjust the mix of free records and free blocks, reserve blocks, and reserve index blocks.

As a general approach, estimate not only the number of inserts but also their distribution throughout the data set. For example, consider a data set with 5 records per block, and 10 data blocks per cluster.[4] Suppose that the data set consists of 300 base records and 200 inserts.

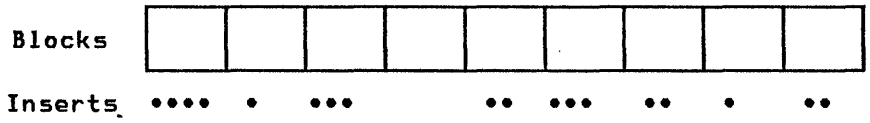
If the inserts are distributed evenly throughout the data set, the pattern of inserts is:



With this kind of distribution you can specify 2 free records per block to absorb the inserts; no free blocks 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 are distributed in this pattern:

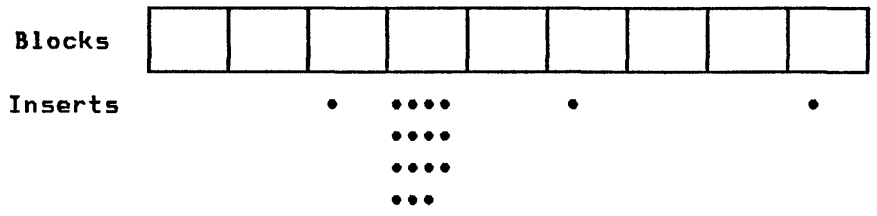


With this distribution you must specify either 3 free records per block, or 20% free blocks with 2 free records per block.

Now suppose the distribution were more uneven:

---

[4] A cluster is a group of blocks; the extent of the group being determined by the structure of the index. This is more fully described in "Data Set Format" on page 70.



In this case a satisfactory mix of free space is 1 free record per block and 40% free blocks.

If the anticipated insert activity is confined to only a few clusters, it is better to use a free pool. A free pool is a group of blocks, at the end of an indexed data set, that are available to be used wherever they are needed within the file. However, in order to use blocks from the free pool, the data set must be structured so that they can be logically connected where they are needed. This structure is specified with the RSVBLK and RSVIX parameters of the define command of the Indexed Access Method utility program.

Use the RSVBLK parameter to indicate the percentage by which a cluster can grow, by taking data blocks from the free pool. The structuring is accomplished by leaving reserve entries in the lowest level index blocks. These reserve entries are not originally used, but can be used later to point to data blocks taken from the free pool.

Use the RSVIX parameter to indicate the percentage by which a cluster grouping can grow by adding new clusters. This structuring is accomplished by leaving reserve entries in the second-level index blocks. These reserve entries are not originally used, but can be used later to point to new lowest-level index blocks taken from the free pool. This lowest-level index block is the seed for a new cluster and can ultimately grow into a full-sized cluster as data blocks are taken from the free pool.

As an example of the advantage of a free pool, assume that a data set contains 50 clusters of 10 data blocks each and 40% of the blocks in the cluster are free blocks. There are 200 free blocks in the data set. If most of the inserts into the data set will fall into a relatively small key range and do not normally require more than 50 blocks, 150 blocks are saved by specifying no free blocks and a 40% RSVBLK.

A 25% FPOOL parameter provides the 50 blocks in the free pool to be used where the inserts are required. The result is that the data set still accepts all the anticipated inserts and 150 blocks are saved.

If insert activity into the data set is anticipated to be relatively even, the space for inserts should be reserved as free records and free blocks. This results in the best response time from the Indexed Access Method.

If, however, insert activity into the data set is to be primarily into one or more areas or key ranges, the space for inserts should be reserved as reserve blocks and/or reserve indexes. This results in the most efficient use of space in the data set.

The space for inserts can be divided between free records, free blocks, reserve blocks, and reserve indexes to suit your requirements.

To determine how many blocks are required for an indexed data set with a given combination of free records, free blocks, reserve blocks, reserve index blocks, and free pool size, use the SE command of the Indexed Access Method utility program (see "Determining Size and Format").

BUILDING THE INDEXED DATA SET

The SE and DF commands of the Indexed Access Method utility program allow you to specify the size and format of your indexed data set and to do the actual data set formatting. Use the SE command to enter those values that determine the size of the indexed data set and to receive size calculation information. Use the DF command to cause the actual data set formatting to occur, using the values previously specified on the SE command.

Determining Size and Format

The design of the data set is determined by these parameters of the SE command:

- BLKSIZE - Block size
- RECSIZE - Record size
- KEYSIZE - Key size
- BASEREC - Estimated number of base records
- FREEREC - Number of free records per block
- FREEBLK - Percentage of free blocks

- RSVBLK - Percentage of reserved data blocks
- RSVIX - Percentage of reserved primary index blocks
- FPOOL - Percentage of free pool
- DELTHR - Percentage delete threshold

The define (DF) command fixes the size of the data set. Therefore, BASEREC, FREEREC, FREEBLK, RSVBLK, RSVIX, and FPOOL should be large enough to accommodate the maximum number of records planned for the data set. To calculate the size of the data set for a given combination of the define parameters, use the SE command.

The DF command allows you to select the immediate write-back option. With this option you can cause modifications to the indexed data set to be written to the file immediately. This contributes to the integrity of the file, but increases response time.

Defining and Creating the Indexed Data Set

The setparms (SE) command allows you to receive the size calculation information without actually performing the data set formatting. The size of the data set and other relevant information are returned to your terminal by the utility. The calculations performed by the setparms function are described in "Data Set Format" on page 70 and are summarized in "Summary of Calculations" on page 84. Use the DF command to actually format the data set. You are prompted for the volume and data set names, and for the immediate write-back option, of the data set to be formatted. (Note: the data set must be previously created using the CR command of the Indexed Access Method utility program and/or the AL command of the $DISKUT1 utility.) This data set is connected and then formatted by the define function. If the data set does not contain sufficient space to support the specified format, the amount of space required is returned to you. Knowing the available space and using the SE command, you can vary the define parameters to design a data set that fits. If the specified data set does not exist, a connect error will occur and you are given the option to retry. If you indicate to retry, you are prompted for the volume and data set names of the data set to be formatted and the function is attempted again.

Connecting and Disconnecting the Indexed Data Set

When you prepare to use an indexed file by the Indexed Access Method, you must issue a LOAD or PROCESS request to connect it to your program.

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

Only one LOAD can be connected to the data set. However, processing can take place concurrently with loading. No LOAD or PROCESS is successful until the file has been formatted using the define command of the Indexed Access Method utility program.

Multiple IACBs can be associated with the same data set. Data integrity is maintained by a locking system that allocates file, record, or block locks to the requesting IACB in order to prevent concurrent modification of index or data records by other requests.

An IACB can hold only one lock at a time; therefore, if your application requires concurrent execution of functions that obtain locks (direct update or sequential update - see "Processing" on page 59 for a description of these functions), you must issue multiple PROCESSes to provide multiple IACBs. The Indexed Access Method retains information about these requests in the IACB.

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

There is no automatic DISCONN on task termination. It is very important that you disconnect your indexed data sets prior to terminating your task; failing to do so may prevent resources allocated to your task from being allocated to other tasks and updates from being written back from the buffer to your data set.

LOADING BASE RECORDS

Base records must be loaded in ascending order by key. Initiate loading of base records with a LOAD request. Then load the records with a PUT for each record. When the desired number of records has been loaded, issue a DISCONN request to terminate the load procedure. The only valid

requests that can follow a LOAD request are:

- PUT
- EXTRACT
- DISCONN

You need not load all the base records at once. A data set that contains some records can be reconnected for loading more records. The key of each new record must be higher than any key already in the data set.

Also, the limit on base records as specified on the define command of the Indexed Access Method utility program cannot be exceeded. If an attempt is made to load a record after the last allocated record area has been filled, an end-of-file condition is returned.

Only one user can have a data set connected for LOAD at any time. Other processing requests can concurrently be made to a data set that is being loaded. However, an attempt to retrieve a record from a data block being loaded can result in a no-record-found condition. You can load an indexed file from a sequential data set by using the load command (LO) of the Indexed Access Method utility program

PROCESSING

Initiate the processing of an indexed data set with a PROCESS request. After the PROCESS, any of the following functions can be requested. Note that the update functions require more than one request.

- Direct reading - Retrieval of a single record independent of any previous request
- Sequential reading - Retrieval of the next logical record from the point of the previous request
- Direct updating - Retrieval of a single record for the purpose of updating that record; subsequent completion of the update by replacing or deleting the original record.
- Sequential updating - Retrieval of the next logical record for the purpose of updating that record; subsequent completion of the update by replacing or deleting the original record

- Inserting - Placement of a single record in its logical key sequence in the indexed data set

- Deleting - Direct removal of a single record from the indexed data set

- Extracting - extract data describing the data set

When the function is complete, another function can be requested, except that a sequential function can be followed only by another sequential function until the sequence is ended. At any time, you terminate the processing by issuing a DISCONN.

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 KLEN (key length) specification.

The record retrieved is the first record in the data set that satisfies the search argument defined by the KEY, KLEN, and KREL (key relation) parameters. The key field is updated to reflect the key in the record that satisfied the search.

If KLEN is specified as less than full key length, only part of the key field is used for comparison when searching the data set. For example, suppose the keys in the data set are AAA, AAB, ABA, and ABB and suppose the key field contains AB0 and KREL (key relation) is EQ.

If KLEN is zero, the search argument is the full key (default) AB0 and a 'record not found' code is returned. If the KLEN specification is 2, the search argument is AB, and the third record is read. If the KLEN specification is 1, the search argument is A, and the first record is read.

Direct Updating

To update a record using direct access:

1. Retrieve the record with a GET request with one of the update MODE/KREL parameters specified.

2. Modify the record in your buffer as desired. However, do not change the key field in the record. Return the updated record to the data set with a PUTUP request.

Alternatively, you can delete the record with PUTDEL, or leave it unchanged by completing the update with a RELEASE request.

A direct update, like a direct read, requires that the KEY parameter be specified as the address of a field of full key length. This field must not be modified during the update.

The only valid requests, other than DISCONN and EXTRACT, that can follow GET for direct update are the requests that complete the update:

- PUTUP
- PUTDEL
- RELEASE

During the update, the subject record is locked; that is, the record is unavailable to any other request until the update is complete. Even if no other action is taken after the GET, the RELEASE is required to release the lock on the record.

Sequential Reading

Use the GETSEQ request to read a record using sequential access. After a sequence has been initiated by a sequential retrieval, only sequential functions can be requested until the sequence is completed with an end-of-data condition or an ENDSEQ request. At any time in the sequence, processing can be terminated with a DISCONN. Also, the sequence is terminated if any error or warning is returned while in sequential mode.

Starting the Sequence. To begin the sequence with the first record in the data set, set the KEY address to zero. To start the sequence with any other record, specify a search argument as for a direct read.

If you specify the search argument when you start the sequence, the key field is modified to reflect the key found as a result of the retrieval of the first record.

Intermediate Retrievals. After the first retrieval, a GETSEQ retrieves the next sequential record regardless of any KEY, KREL, or KLEN specification. Therefore, you can use the same GETSEQ statement in a loop to read all the records in the sequence. If you specify a search argument on intermediate retrievals, it is ignored and the key field is not modified on intermediate retrievals.

Ending the Sequence. To end the sequence before the end of data is reached, specify ENDSEQ. The sequence is ended automatically at the end of data. An end-of-data condition occurs when an attempt is made to retrieve a record after the last record in the data set.

If you specify the EODEXIT parameter in the PROCESS, it is not necessary to test for the end-of-data return code. When the end-of-data condition occurs, control transfers to your end-of-data routine.

Sequential Updating

To update a record using sequential access, retrieve the record with a GETSEQ request with one of the update MODE/KREL parameters specified. The sequential retrieval for update is the same as the sequential read. A search argument is used only on the first retrieval of a sequence and is not specified if the sequence is to begin with the first record in the data set. The sequence is terminated with an ENDSEQ or with an end-of-data condition.

The sequential update is completed in the same way as a direct update. The key in the record cannot be modified. The record can be returned to the data set with a PUTUP, deleted with a PUTDEL, or left unchanged by specifying RELEASE. When the update is complete, another sequential read or update can be requested.

It is valid to terminate the sequence with ENDSEQ, or terminate processing with a DISCONN either before or after completing the update. Figure 13 summarizes the protocol for sequential processing.

Inserting

To insert a new record in a data set connected for processing, specify PUT. The Indexed Access Method uses the key within the record in your buffer to insert the record in proper key order in the data set.

The key of the inserted record must be different from any key in the data set; otherwise a duplicate key error occurs. The key can be higher than any key in the data set; that is, it is permissible to insert a record with a new high key.

A request for sequential update: GET for update can be followed by:	
1. A request to end processing: DISCONN	
2. An end-of-data condition: automatic end-of-sequence and transfer of control to EODEXIT routine 3. A request to end the sequence: ENDSEQ	End-of-sequence can be followed by: 1. A request to end processing: DISCONN 2. Any processing function: GET, PUT, DELETE
A request to complete the update: 4. PUTUP 5. PUTDEL 6. RELEASE	The completed update can be followed by: 1. A request to end processing: DISCONN 2. A request to end the sequence: ENDSEQ 3. A sequential request: GETSEQ

Figure 13. Protocol for Sequential Updating

If there is no free space in the area associated with the insert or no blocks in the free pool, a no-more-space condition occurs. The no-more-space condition does not mean the data set is full; there can be free space in unrelated blocks. It indicates a need for data set reorganization. This procedure is described in "Reorganization" on page 69.

Deleting

Use DELETE to delete a record from the data set. The full key of the record must be specified. If there is no record with that key, a negative return code gives a warning.

Deletion is also performed as part of updating by following a GET for update with a PUTDEL request.

Extracting

You can extract information about a data set from the file control block (FCB). This includes information such as key length, key displacement, block size, record size, and other detailed data regarding the data set structure.

This data is requested by EXTRACT. Execution of this function causes copying of the file control block to the specified user area. You must provide the storage into which the data is copied. The data set must be connected by LOAD or PROCESS.

HANDLING ERRORS

All executed Indexed Access Method requests return a signed number, called a return code, in the task code word (referred to by task name) of the TCB. The return code reflects the condition of the requested function. Return codes are grouped in three categories:

- -1 - Successful completion
- Positive - Error
- Negative - Warning

Error Exit Routine

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 this request or any subsequent Indexed Access Method request for the duration of this PROCESS or LOAD terminates with a positive return code.

If the address of an error exit routine is not specified, the next sequential instruction after the request is executed regardless of the value of the return code.

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 reserved by the PROCESS or

LOAD request. This is used by any subsequent request until a DISCONN is issued.

For example, GET uses the read function. If the read terminates with a positive return code, that return code is saved in the location reserved for the system function return code in the PROCESS associated with the GET. The GET also terminates with a positive return code in the task control word that indicates a read error. The cause of the read error is determined from the system function return code.

Data Set Shut Down

Sometimes an I/O error that is not associated with a specific request occurs. For example, Task A issues a GET on data set X. In order to secure buffer space, the Indexed Access Method writes out a block to data set Y and, in the process, an error occurs. Data set Y is damaged but there is no requesting program to accept an error return code.

The error is recorded by setting the data-set-shut-down condition for data set Y. When this condition exists, no requests other than a DISCONN are accepted for this data set.

Later, if Task B issues a GET on data set Y, the request is terminated with a data-set-shut-down return code. Task B should issue a DISCONN and use recovery procedures to reconstruct the data set. An initial program load (IPL) cancels the data-set-shut-down condition.

Deadlocks

Since the Indexed Access Method uses record and block locks to preserve file integrity, deadlock conditions are possible. A deadlock is a condition where two or more tasks interact in such a way that one or more resources become permanently locked and further progress is not possible.

A deadlock can also occur when two IACBs from the same task request a lock on the same record or a lock on the same block in sequential mode.

In this section, the term deadlock refers not only to a true deadlock, but also to an apparent deadlock, in which a task holds a resource for an unreasonably long period of time.

Application tasks should not use the Indexed Access Method in such a way that a record or block remains locked for a long period of time, since 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, then wait for the operator to modify it before completing the update.
- Do not retrieve a record in sequential mode, display the record at the terminal, then wait for an operator response before continuing the sequential operation.

In both of these cases, a record or block is locked during operator "think" time and could be held for minutes or hours.

Every effort should be made to avoid a deadlock situation. There is no way to break a deadlock except to release the locks that are being waited on. Even terminating the tasks involved might not break the deadlock because termination processing cannot occur until any outstanding requests issued by a terminating task have completed.

EXECUTING THE APPLICATION PROGRAM

Application programs that use the Indexed Access Method are executed the same as other application programs. Because the Indexed Access Method and the indexed data sets are resources available to all tasks, delays can occur under heavy system usage. Specifically, with more than one task using the Indexed Access Method there can be contention between tasks for any of these resources:

- Entire indexed file
- Index block in the data set
- Data block in the data set
- Data record in the data set
- Buffer space from the system buffer pool

For example, during the execution of a request from Task A, some buffer space can be required and an index block or data block or record can be locked (made unavailable to other requests). If a request from Task B requires more buffer space than remains available or if it requires a

locked block or record, that request is delayed until the required resource is freed.

In general, resources required by the Indexed Access Method are allocated only for the duration of that request. There are two exceptions:

- 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, PUTDEL, RELEASE, or by a 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; sequential processing is fast because no search is required. When a GET requires a record from the next block, the current block and buffer are released. Pending requests for 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, one block and buffer are locked for the entire sequence. The sequence is terminated by an end-of-data condition, by an ENDSEQ, or by a DISCONN.

The update and the sequence should be completed promptly. Use the following guidelines to avoid problems with these resources.

1. Disconnect all indexed data sets before task termination. The DISCONN releases locked records or blocks and writes out buffers for that data set that have not already been written.

2. With multiple Indexed Access Method users on the system, use direct access rather than sequential access to retrieve a sequence of records interactively. A suggested method is:

   a. Retrieve the first record by key.

   b. Extract the key from that record and save it for the next retrieval.

   c. Retrieve the next record using the saved key and a greater-than key relation.

   d. Repeat the second and third steps for the duration of the sequence.

MAINTAINING THE INDEXED DATA SET

The Indexed Access Method does not provide specific programs to perform indexed data set backup and recovery, nor does it include services to delete the data set or dump to the printer. These procedures are readily provided by a combination of EDX and Indexed Access Method services as suggested below. The Indexed Access Method utility program provides services to help you reorganize your data set as described below.

Backup and Recovery

To protect against the destruction of data, at regular intervals you should make a copy of the indexed data set (or the logical volume in which the data set exists) using the system COPY utility. During the interval between copies, you should keep a journal file of all transactions made against the indexed data set.

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

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

If a data set shut down condition exists, IPL again. Then issue a PROCESS to the REPORT data set and, using the journal file, recreate the transactions that occurred since the backup copy was made.

Recovery Without Backup

If you do not use the backup procedures outlined above and you encounter a problem with your data set, you still may be able to recreate your file. However, the status of requests made prior to the problem is uncertain.

To recreate your data set, follow the steps given below as a method of reorganizing your data set. After recreating the data set, verify the status of requests made at the time the problem occurred.

Reorganization

An indexed data set must be reorganized when required inserts fail because of lack of space. This condition does not imply that there is no more space in the data set; it means that there is no space in the area where inserts must be made. Therefore, you can reorganize without increasing the size. The following steps provide a method of performing a reorganization:

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

2. Use the define command (DF) of the Indexed Access Method utility program to define a new indexed data set. Carefully estimate the number of base records and the amount and mix of free space in order to minimize the need for future reorganizations. Guidelines for making these estimates appear at the beginning of this chapter.

3. Use the reorganize command (RO) of the Indexed Access Method utility program to load the new indexed data set from the indexed data set to be reorganized.

   Alternatively, you can use the unload command (UN) of the Indexed Access Method utility program to transfer the data from an indexed data set to a sequential data set, then use the load command (LO) to load it back into the indexed data set. The result is a reorganized indexed data set.

4. Use system utilities to perform any desired housekeeping operations. The old data set can be deleted, and the new data set can subsequently be renamed to the name of the old data set.

Dumping

To print user records, retrieve the records sequentially and print them. Use the DP command of the $DISKUT2 utility for a hexadecimal dump of the entire data set including control information, index blocks, and data blocks. Information on the $DISKUT2 utility can be found in the Utilities, Operator Commands and Program Preparation manual.

Deleting

Delete an indexed data set in the same manner in which you delete any data set. From your terminal, use the DE command of the $DISKUT1 utility (see the Utilities, Operator Commands and Program Preparation manual).

DATA SET FORMAT

The define command of the Indexed Access Method utility program formats and creates the indexed data set. The information required to determine the format and the number of blocks in the data set is provided by ten parameters. These parameters are specified using the setparms command. Examples in this chapter use the following values for the parameters:

Parameter Name	Value Addressed by Parameter
BLKSIZE	Block size = 256 bytes
RECSIZE	Record size = 80 bytes
KEYSIZE	Key size = 28 bytes
BASEREC	Number of base records = 1000
FREEREC	Number of free records per block = 1
FREEBLK	Percentage of free blocks = 10
RSVBLK	Percentage of reserved blocks = 10
RSVIX	Percentage of reserved index = 10
FPOOL	Percentage of free pool = 50
DELTHR	Percentage of blocks to retain when deleting records; this value is allowed to default.

BLOCKS

The indexed data set is composed of a number of fixed length blocks. The block is the unit of data transferred by the Indexed Access Method, and is a multiple of 256. A block is addressed by its relative block number (RBN) such that the first block in the data set is located at RBN 0.

Note that the RBN is a block number used only in indexed data sets, by the Indexed Access Method. A block as used in the Indexed Access Method differs from an EDX 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 bytes.

2. The RBN of the first block in an indexed data set is 0. The record number of the first EDX record in a data set is 1.

The size, in 256-byte records, of the data set is calculated by the define command of the Indexed Access Method utility program and returned to the terminal.

There are three kinds of blocks in an indexed data set: a file control block (FCB), index blocks, and data blocks. These blocks are all the same length, as defined by BLKSIZE, but they contain different kinds of information: control information, index entries, and data records. The control information is contained in block headers and the FCB; for a description of control information, see Figure 14. Figure 14 shows examples of the three block types.

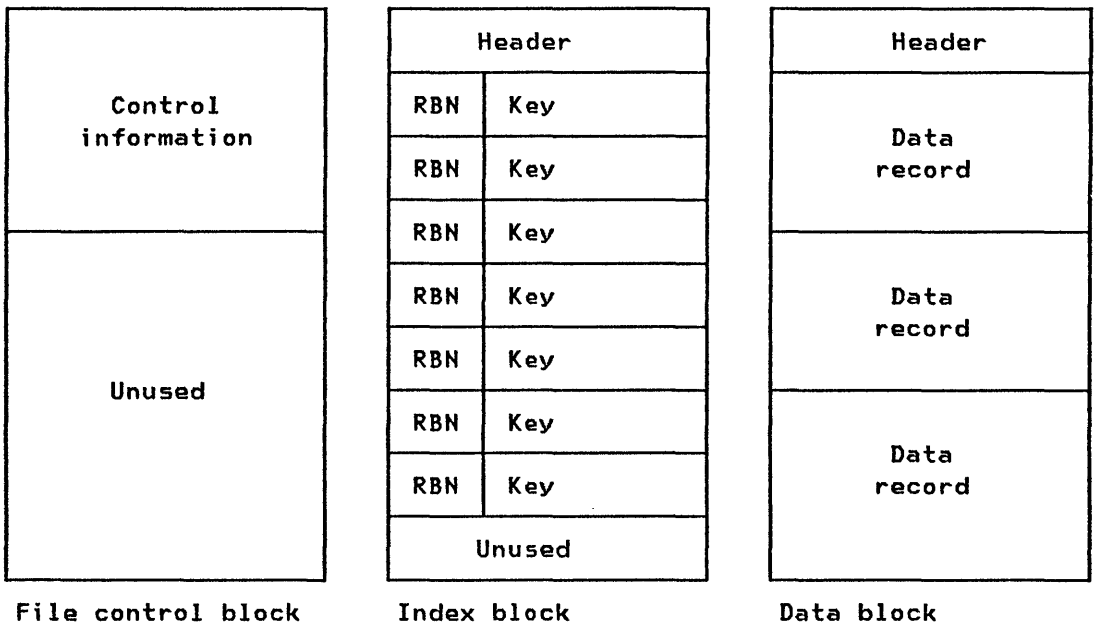


Figure 14. Indexed Data Set Block Types

FCB

The file control block (FCB) is the first block in the data set (RBN 0). It contains a fixed amount of control information.

Index Block

An index block contains a header followed by a number of index entries. Each index entry is a key-pointer pair. The key is the highest key associated with a 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 key size is odd then the field is padded with one byte to make key entry 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 size 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.

Data Block

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

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

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, there can be 3 records in a data block. In this example, there is no unused area. The key field of the last record slot in a data block indicates the high key for the data block. If all records of the data block are not currently used, the key field of the last record slot is normally the same as the key field of the last used record in the block. However, it is pos-

sible for the key field of the last record slot to contain a key higher than that of any record in the block. This can occur if the last record of the block has been deleted. Thus, deletion of a record does not reduce the key range for the block.

The Index

The index of an indexed data set 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 prime index blocks. At the second level are index blocks containing entries that point to the prime index blocks. There are enough levels so that the highest level consists of a single index block.

Prime Index Blocks

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

- Allocated (used) entry. This type of entry points to an active data block. The key portion of the entry is initialized to binary 1's (all bits on). The key portion of the entry 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 the index block. The number of index entries initially allocated when the indexed data set is loaded is calculated as 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) (see Figure 15).

- Free block entry. This type of 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 initially formatted as free entries when the indexed data set is loaded is the specified percentage (FREEBLK) of the total number of entries, with the result rounded up if there is a remainder.

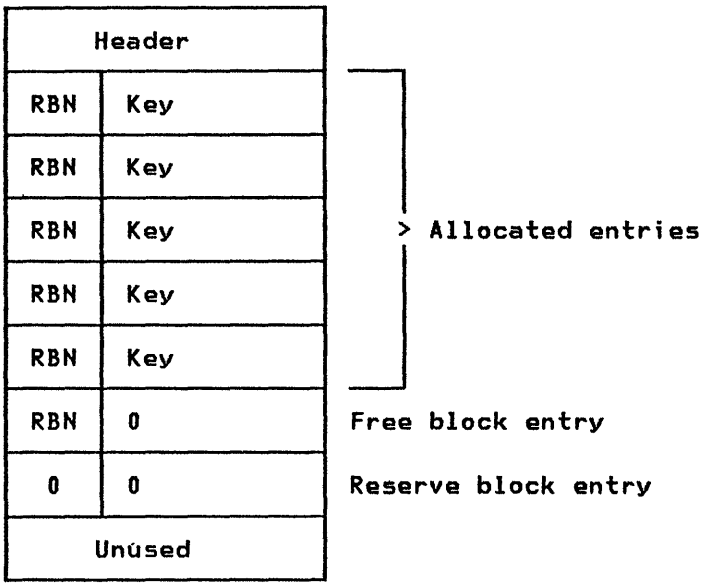


Figure 15. Example of Prime Index Block

- Reserve block entry. This type of entry does not point to a block. It 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. 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 and the number of reserve block entries together require all index entries, the number of reserve block entries is reduced by 1. This provides at least one allocated entry per index block.

In order to calculate the number of prime index blocks in an indexed data set, you must first know the initial number of data blocks allocated in the indexed data set. The initial number of data blocks is calculated as the specified number of base records (BASEREC) divided by the number of allocated (not free) records in a data block, with the result rounded up if there is a remainder. The number of prime index blocks can then be calculated as the initial number of allocated data blocks divided by the number of allocated entries per prime index block, with the result rounded up if there is a remainder.

Second-Level Index Block

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

- Allocated (used) entry. This type of entry points to an existing prime index block. The key portion of the entry is initialized to binary 1's (all bits on). The key portion of the entry contains the highest key from the prime index block. The pointer portion contains the RBN of the prime index block. Allocated entries are the first entries in the index block. The number of index entries initially allocated when the indexed data set is loaded is calculated as the total number of entries per index block, less the number of reserve index entries.

- Reserve index entry. This type of entry does not point to a block. It is reserved for later use as a pointer to a prime 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 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. This provides at least one allocated entry per second-level index block.

The number of second-level index blocks is calculated as the number of prime index blocks divided by the number of allocated entries per second-level index block, with the result rounded up if there is a remainder (see Figure 16).

Higher Level 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 (used) entries. The key portion of the entry contains the highest key from the next lower level index block. The pointer portion contains the RBN of the next lower level index block. The number of blocks at any higher index level is calculated as the

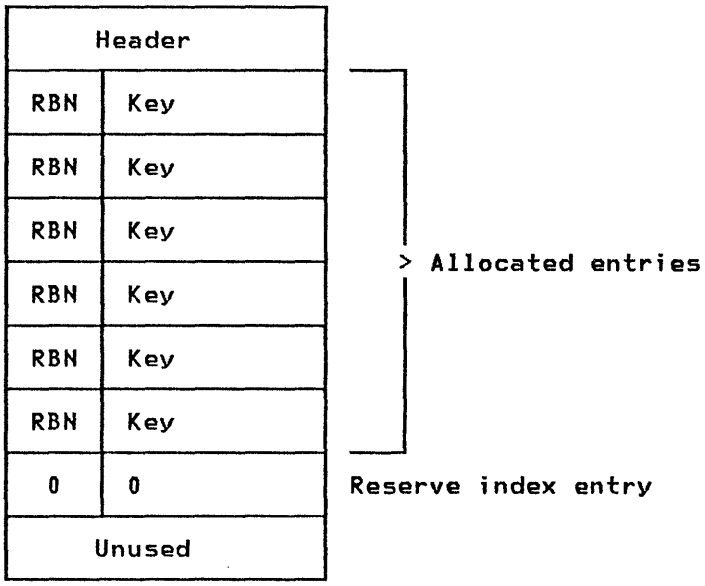


Figure 16. Example of Second-Level Index Block

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 (see Figure 17).

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 supporting 17 levels of index, any given indexed data set is formatted with only as many index levels as are required for the specified number of records. If an indexed data set has not been fully loaded, it is possible that one or more higher index levels are not yet required, even though they exist in the file structure. In this case, the unnecessary higher levels are not used.

Index Example

In the sample data set described at the beginning of this chapter, 500 data blocks are initially allocated to the data set. Each prime index block contains one free block entry, one reserve block entry, and five allocated entries; therefore, the total number of prime 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

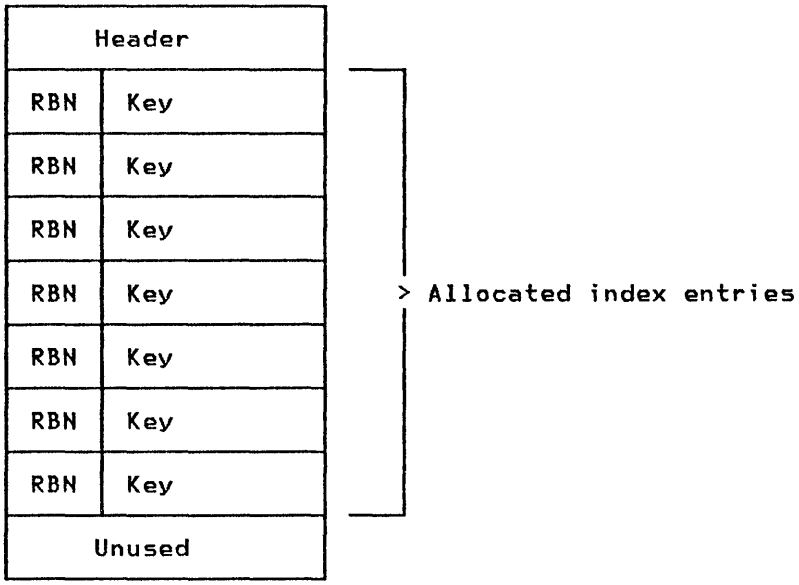


Figure 17. Example of Higher-Level Index Block

entries in higher level index blocks is seven. This results in three index blocks at the third level and one at the fourth level.

Therefore the sample data set contains a total of 121 index blocks. Of these blocks, 100 are prime index and the remaining 21 are high-level index blocks. This distinction is important because, as shown later in this chapter, high-level index blocks are located contiguously at the beginning of the data set (after the FCB), while prime index blocks are scattered throughout the file with the data blocks. Figure 18 shows the structure of the high-level index blocks.

Cluster

Data records are loaded into the data blocks in ascending order by key. Each data block is pointed to by a prime index block entry that contains the high key of the data block (that is, the key from the last record slot in the data block).

Prime index blocks and data blocks are stored together in the data set in groups called clusters. Each cluster begins with a prime index block followed by as many data

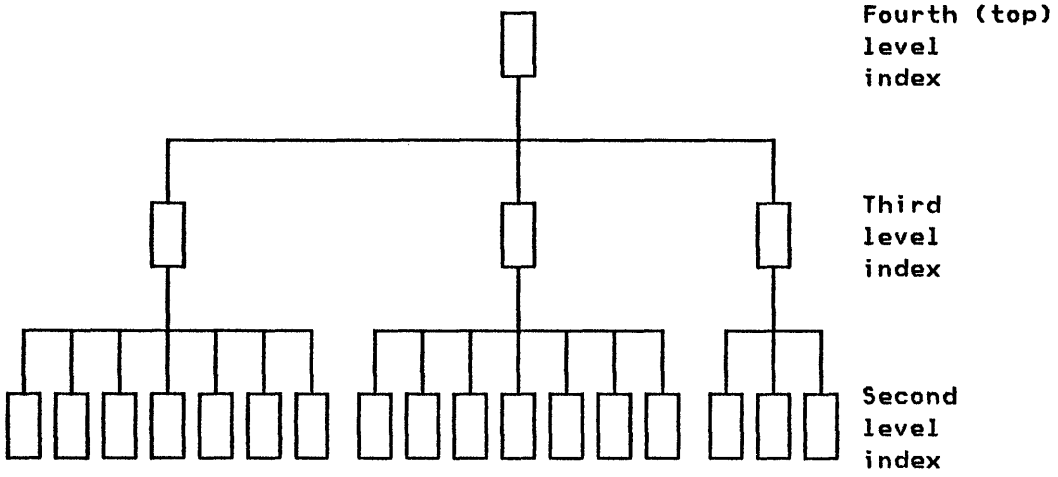


Figure 18. High-Level Index Block Structure

blocks as there are allocated or free entries in the index block. Data blocks can be of two types: allocated data blocks and free data blocks. The prime index block can also contain unused slots for reserved blocks. For example, if there are seven entries in an index block, there are eight blocks in a cluster: one prime index block followed by up to 7 data blocks. If there are reserve blocks specified, even though the cluster can expand to eight blocks, 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, then there would only be seven blocks in the cluster initially (one index block and six data blocks). This is illustrated by the following diagram.

Prime index block	Data block	Data block	Data block	Data block	Data block	Data block

FREE SPACE

When an indexed data set is loaded with data 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 blocks.

Free Records

Free records are areas reserved at the end of each data block. The FREEREC parameter of define command of the Indexed Access Method utility program 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. See Figure 19.

When records are loaded, the allocated records are filled, and the free records are skipped over. During processing, a record can be inserted in a block that contains a free record.

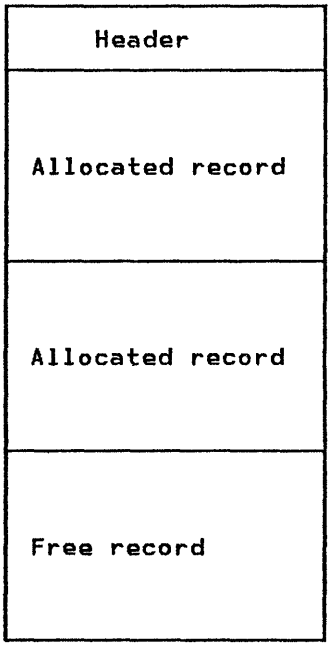


Figure 19. Example of a data block

Free Blocks

Free blocks follow the allocated data blocks within each cluster. For example, if the cluster contains six data blocks and you specify 10 as the percentage of free blocks, then there are five allocated blocks and one free block in each cluster.

Prime index block	Alloc. data block	Alloc. data block	Alloc. data block	Alloc. data block	Alloc. data block	Free data block

When records are loaded, the allocated record areas in the allocated data blocks are filled, and the free blocks are skipped over. During processing, as data blocks become full, a free block can be included in the logical block sequence to provide space for more insertions.

Reserve Blocks

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, provided the prime index block contains a reserve block entry. In this case, the reserve block entry in the prime index block points to the block, and the data block becomes a normal used data block.

Reserve Index Entries

Reserve index entries in second-level index blocks allow the index structure to be expanded by adding new prime index blocks. These, in turn, can have data blocks associated with them, thus forming new clusters. This process of forming a new cluster is sometimes called a cluster split.

The initial number of allocated data blocks in the data set can be calculated as the specified number of base records (BASEREC) divided by the number of allocated (not free) records in a data block, with the result rounded up if there is a remainder.

The number of clusters in the data set can be calculated as the initial number of allocated data blocks divided by the number of allocated entries in each prime index block, with the result rounded up if there is a remainder.

The total number of free blocks in the data set (not including the free pool) is calculated as the number of clusters in the data set multiplied by the number of free entries in each prime index block.

The Last Cluster

The last cluster in the data set 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.

For example, suppose you intend to load 1000 records in an indexed data set 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 data set 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. The number of clusters in a data set is:

$$\frac{\text{number of allocated blocks}}{\text{number of allocated blocks per cluster}}$$

The number of clusters in this example is 500/5 or 100 clusters. Note that in both these calculations, if the quotient is not an integer, it must be rounded up (rather than truncated) in order to accommodate all of the base records. Thus, in the last allocated block, there can be a few more allocated records than required. However, the last cluster can be a short cluster because it will have only the required number of allocated blocks.

In this example, the number of allocated blocks divided by the number of allocated blocks per cluster (500/5) equals 100 with no remainder. If there is remainder it represents the number of allocated entries in the prime index block for the last cluster, thus the number of allocated data blocks in that cluster. Unused entries in the last prime index block are treated as reserve block entries.

Sequential Chaining

Data blocks in an indexed data set are chained together by forward pointers located in the headers of data blocks.

Only allocated data blocks are included in this sequential chain. Free blocks are skipped over. This provides efficient sequential processing of the data set with no need to reference the index. When a free block is converted to an allocated block, it is included in the chain.

Free Pool

If you specify that you want a free pool (by the FPOOL parameter of define command of the Indexed Access Method utility program), your indexed data set contains a pool of free blocks at the end. 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 chain to become either a data block or a prime index block. When this is done, the block is taken from the beginning of the chain, and its address (RBN) is placed in the appropriate prime 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 prime index block). Any block in the free pool can be used as either a data block or as a prime index block.

When a data block becomes empty because of record deletions, it is sometimes possible to return it to the free pool (depending on the delete threshold (DELTHR) parameter). When this is done, reference to the block is removed from the prime index block, and the block is placed on the beginning of the free pool chain. Index blocks are never returned to the free pool.

The calculation of the initial size of the free pool consists of several steps, as follows:

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

- Each reserve index entry in a second-level index block represents a possible use of a prime index block from the free pool. The number of prime index blocks that can be assigned from the free pool is the number of second-level index blocks times the number of reserve index entries in each second-level index block.

- Each prime index block taken from the free pool consists entirely of empty (reserve block) entries. New data blocks can then be taken from the free pool for these entries in the new prime index block. The total number of such data blocks is the total number of entries per index block times the number of new prime index blocks (calculated in the previous step).

- The maximum number of blocks that can conceivably be taken from the free pool is the sum of the above three calculations. This is the maximum possible free pool.

- The actual number of blocks in the free pool is the specified percentage (FPOOL) of the maximum possible free pool, with the result rounded up if there is a remainder.

STORAGE AND PERFORMANCE INFORMATION

STORAGE REQUIREMENTS

The minimum amount of storage required by the Indexed Access Method to perform all functions is about 14KB, not including the link module or the user error exit routine. This is based on the following assumptions:

- A maximum block size of 256 bytes for any indexed data set. Since the buffer must be large enough for two blocks, we have assumed a 512 byte buffer. If your maximum block size is larger, you should subtract the 512 and add in double your block size. You can improve performance by making the buffer larger.

- One user connected to an indexed file at a time. If you will have more than one user connected, you should add about 625 bytes per additional user.

The IBM-supplied link module is included in your application program (see Figure 12). Its size is about 250 bytes.

INDEXED FILE SIZE

The structure of an indexed file is highly dependent on parameters you specify when you create the file. This is described in "Data Set Format" on page 70. The calculations presented there are summarized in "Summary of Calculations" on page 84.

PERFORMANCE INFORMATION

Performance of the Indexed Access Method is primarily determined by the structure of the indexed data set being used. This structure is determined by parameters you specify when you create the data set. This is described in "Data Set Format" on page 70. Performance is affected by file structure in the following ways:

- File size. A large file spans more cylinders of the direct access device, so the average seek to get the the record you want is longer.

- Number of index levels. A file with many index levels requires more accesses to get to the desired data record, thus degrades performance. Factors which influence the number of index levels are:

  - Number of records in data set.

  - Amount and type of free space.

  - Block size.

  - Key size.

  - Data record size.

To get a feeling for the affect of the various parameters on the file structure, you should calculate several examples. Refer to "Summary of Calculations."

In addition to file structure, the following factors also influence performance:

- Buffer size. 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 data set.

- Contention. If many tasks are concurrently using the Indexed Access Method, interference can result, and performance is degraded.

SUMMARY OF CALCULATIONS

The following calculations are used to structure an indexed data set. In the calculations requiring division, results with non-zero remainders are either:

truncated [T], or rounded up [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 [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 minus 16, divided by index entry size; result truncated.

[5] = (BLKSIZE-16) / [4] [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] [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 PIXB require all PIXB entries, subtract one from reserve entries per PIXB.

[7] = RSVBLK % of [5] [U]

(-1 if [6] + [7] = [5])

8	Allocated entries per PIXB = total entries per index block minus free entries per PIXB, minus reserve entries per PIXB.	8 = 5 - 6 - 7

Index Block (SIXB)

9	Reserve entries per secondary index block (SIXB) = specified percentage of total entries per index block; result rounded up. If reserve entries per SIXB require all SIXB entries, subtract one.	9 = RSVIX % of 5 U (-1 if 9 = 5)
10	Allocated entries per SIXB = total entries per index block minus reserve entries per SIXB.	10 = 5 - 9

Delete Threshold

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 cluster = allocated entries per PIXB, plus one-half of free entries per PIXB; result rounded up.

11 = 8 + 6 / 2 U

or

c. If the RSVBLK parameter was specified, and the DELTHR parameter was specified:
Number of blocks to retain in 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 U
	(If 0, set 11 to 1)

Data in Data Set

12	Initial allocated data blocks = base records divided by allocated records per data block; result rounded up.	12 = BASEREC / 3 U
13	Number of clusters in data set = initial allocated data blocks, divided by allocated entries per PIXB; result rounded up.	13 = 12 / 8 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	Number of secondary index blocks (SIXBs) = number of PIXBs, divided by allocated entries per SIXB; result rounded up.	16 = 15 / 10 U

17	Calculate 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 $_{i}$ = 17 $_{i-1}$ / 5 U
18	Total number of index blocks = sum of index blocks at each level until a level containing a single index block is attained.	18 = 15 + 16 + (Sum of all 17 s)

Free Pool

19	Number of new data blocks that can be assigned to existing clusters = reserve entries per PIXB, times number of PIXBs.	19 = 7 * 15
20	Number of new clusters (PIXBs) that can be created = reserve entries per SIXB, times number of SIXBs.	20 = 9 * 16
21	Number of new data blocks that can be assigned to new clusters = total entries per index block, times number of new clusters that can be created.	21 = 5 * 20

22	Maximum possible free pool = number of new data blocks that can be assigned to existing clusters, plus number of new clusters (PIXBs) that can be created, plus number of new data blocks that 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 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, plus total number of free blocks in data set, plus actual number of free pool blocks.	24 = 1 + 18 + 12 + 14 + 23

$

A

B

C

D

J

K

L

M

N

O

P

Q

R

S

T

U

V

READER'S COMMENT FORM

GC34-0328-0

IBM Series/1 Event Driven Executive
Version 1.1 Planning 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

Cut or Fold Along Line

Please indicate your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.
(Elsewhere, an IBM office or representative will be happy to forward your comments.)

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
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

International Business Machines Corporation
General Systems Division
4111 Northside Parkway N.W.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
(International)

GC34-0328-0
Printed in U.S.A.

READER'S COMMENT FORM

GC34-0328-0

IBM Series/1 Event Driven Executive
Version 1.1 Planning 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

Cut or Fold Along Line

Please indicate your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

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
Systems Publications, Dept 27T
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

International Business Machines Corporation
General Systems Division
4111 Northside Parkway N.W.
P.O. Box 2150, Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
(International)

GC34-0328-0
Printed in U.S.A.

International Business Machines Corporation

General Systems Division
4111 Northside Parkway N.W.
P. O. Box 2150
Atlanta, Georgia 30301
(U.S.A. only)

General Business Group/International
44 South Broadway
White Plains, New York 10601
(International)

GC34-0328-0
Printed in U.S.A.