IBM Technical Newsletter

This Newsletter No.	SN34-0685
Date	January 22, 1981
Base Publication No.	SC34-0312-2
File No.	S1-34
Previous Newsletters	None

IBM Series/1
Event Driven Executive
System Guide
Program Numbers: **5719-XS1 5719-XS2 5719-MS1**
5719-XX2 5719-XX3 5719-AM3
5719-UT3 5719-UT4
5719-LM5 5719-LM6
5719-LM2 5719-LM3

This Technical Newsletter provides replacement pages for the subject publication. Pages to be inserted and/or removed are:

9, 10	123, 124	191.1, 191.2 (added)
23, 24	127, 128	192
57, 58	128.1, 128.2 (added)	193
73, 74	139, 140	193.1, 193.2 (added)
77 through 80	145 through 148	194
87, 88	148.1, 148.2 (added)	194.1, 194.2 (added)
93 through 98	151 through 156	203 through 206
99	161	221, 222
99.1, 99.2 (added)	161.1, 161.2 (added)	269, 270
100	162	295, 296
101, 102	165, 166	297
103	167	297.1, 297.2 (added)
103.1, 103.2 (added)	167.1, 167.2 (added)	298
104	168	299, 300
107 through 110	173, 174	307, 308
115	187	309
115.1, 115.2 (added)	187.1, 187.2 (added)	309.1, 309.2 (added)
116	188	310
116.1, 116.2 (added)	189, 190	319 through 326
117, 118	191	341, 342

A technical change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

Corrections and editorial changes have been made throughout this book. These changes are identifiable by a vertical bar to the left of the change.

Note. Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432

Printed in U.S.A.

Sort/Merge

The Sort/Merge licensed program sorts and merges records from up to eight input data sets into one output data set in either ascending or descending order. You can specify one or more control fields in the records to be sorted. The Sort/Merge program compares the control fields to determine the relative sequence of the records.

The Event Driven Executive Sort/Merge program executes under the Basic Supervisor and Emulator.

Publications:

- IBM Series/1 Event Driven Executive Sort/Merge: Programmer's Guide, SL23-0016
- IBM Series/1 Event Driven Executive Sort/Merge: Specifications Sheet Form, GX23-0009

Series/1 Macro Assembler

The Macro Assembler converts text data sets containing machine, assembler, and macro instructions that have been coded in the Series/1 instruction set into object modules. The object modules can then be processed by the linkage editor.

When the assembler is used in conjunction with the Macro Library, applications coded in the Event Driven Language can also be processed by the Macro Assembler, including customizing the supervisor. You can also include in the macro library your own macros for commonly used routines. The Macro Assembler and the Macro Library can be used in place of the Program Preparation Facility ($EDXASM).

With the Macro Assembler you can assemble device support modules or modules that modify supervisor functions. You can also assemble exit routines written in Series/1 Macro Assembler language. The resulting object module is input to the Program Preparation Facility linkage editor, together with your applications generated in Event Driven Language instructions, PL/I, FORTRAN IV, and/or COBOL. Your program will execute under the Basic Supervisor and Emulator after it has been processed by the library update utility ($UPDATE).

Publications:

- IBM Series/1 Event Driven Executive Macro Assembler, GC34-0317
- IBM Series/1 Macro Assembler Reference Summary, SX34-0076

Multiple Terminal Manager

The IBM Series/1 Event Driven Executive Multiple Terminal Manager provides a set of high level functions that simplify the design, implementation, and maintenance of transaction-oriented applications. Programs written in COBOL, PL/I, FORTRAN IV, or Event Driven Language can execute in an interactive environment, where one or more applications can run concurrently using one or more display devices. Additional interfaces are provided for indexed or direct files (access to indexed files requires the Indexed Access Method). An operator interface for functions such as sign on, connect or disconnect, terminal status reports, and listing the screens and programs available are also provided.

Publications: Refer to the Multiple Terminal Manager topics in the master index of this publication.

Indexed Access Method

The Indexed Access Method provides data management facilities that support indexed file operations. It allows you to build, access, and maintain records in indexed data sets via a predetermined field called a key. An index of keys provides fast access to records in an indexed data set. The access method supports a high degree of insert/delete activity, providing both direct and sequential access to the data from multiple, concurrently executing programs. Applications that use the Indexed Access Method can be programmed in the Event Driven Language, PL/I, or in COBOL. It is supported by the Sort/Merge licensed program, which will accept Indexed Access Method data sets as input files. Also provided are utilities to define and maintain indexed data sets.

The Indexed Access Method provides keyed access to data to support a variety of applications, ranging from batch processing to interactive applications.

The data file organization provides direct and sequential processing of files. This is accomplished by using cascading index techniques for direct processing and by sequence chaining of the data blocks for sequential processing.

The access method supports files which have high add/delete activity (such as open order files) with nominal performance degradation. This is accomplished by distributing free space for additions throughout the file, by updating and inserting additions in place, and by dynamically reclaiming space after deletions.

- 4979 Display Station
- 3101 Display Terminal or equivalent teletypewriter device

Minimum Licensed Program Requirements

The programs you require depend upon your application and which language you will use to code your applications. The choices are COBOL, FORTRAN IV, PL/I, Event Driven Language, or Macro Assembler Language.

The first requirement is the Basic Supervisor and Emulator. Then, based upon your choice of languages and your type of work, the following can be used as guidelines:

- COBOL

  Program preparation requires the COBOL Compiler and Resident Library, the Utilities, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler. It allows you to:

  - Install the COBOL Compiler and Resident Library and the COBOL Transient Library
  - Allocate data sets
  - Enter source programs
  - Compile
  - Link edit

  Execution and test require the COBOL Transient Library and the Utilities. During execution and test, you may:

  - Use diagnostic aids
  - Load programs
  - Back up and copy data sets

- PL/I

  Program preparation requires the PL/I Compiler and Resident Library, the Utilities, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler; it allows you to:

- Install the PL/I Compiler and Resident Library and the PL/I Transient Library
- Allocate data sets
- Enter source programs
- Compile
- Link edit

Execution and test require the PL/I Transient Library and the Utilities. During execution and test, you may:

- Use diagnostic aids
- Load programs
- Back up and copy data sets

• FORTRAN IV

Program preparation requires FORTRAN IV, the Utilities, the Mathematical and Functional Subroutine Library, and the link editor of either the Program Preparation Facility or the Series/1 Macro Assembler; it allows you to:

- Install FORTRAN IV and the Mathematical and Functional Subroutine Library
- Allocate data sets
- Enter source programs
- Compile
- Link edit

Execution and test require the the Utilities. During execution and test, you may:

- Use diagnostic aids
- Load programs
- Back up and copy data sets

• Event Driven Language

3. When a program is loaded by the $L operator command

4. During execution of some system utility programs

A general data set specification consists of two parts:

1. The data set name (dsname)

2. An optional volume label (volume) which specifies the volume on which the data set resides

The format for a data set specification is:

```
dsname,volume
```

The volume specification is optional and if not specified, the system assumes that the target data set resides on the primary volume on the direct access device from which the system was IPLed.

dsname An alphameric character string of eight characters. When fewer than eight characters are specified, blanks are added to the string.

volume An alphameric character string of six characters. To locate the volume on a disk, it must have been defined in the VOLSER= parameter of a DISK configuration statement in the system I/O definition. To locate the volume on a diskette or tape, the TAPE or DISK statement must be in the system I/O definition and the volume name loaded into the system by issuing the operator command $VARYON, specifying the diskette or tape device address. The diskette must have been initialized by $INITDSK. Tapes must be initialized by the $TAPEUT1 utility. When fewer than six characters are specified, blanks are added to the right to complete the string.

Two special data set names are known to the system and must be used with care:

$$EDXVOL Used to obtain absolute record reference to an entire volume on disk or diskette.

$$EDXLIB Used to obtain absolute record reference to the beginning of the volume directory on disk or diskette within a volume.

Note: Errors may occur if either of these two special data set names are used to refer to deleted or uninitialized HDR1 (Basic Exchange) records.

STORAGE CAPACITIES

Disk/Diskette

The following table summarizes storage capacities of the various Series/1 direct access storage devices.

Device	Storage capacity (records)	Cyl/dev	Logical rcds/trk	Trk/cyl	Volume max (cyls)
Single-sided (type 1) diskette	949	77*	13	1	73
Double-sided (type 2) diskette	1924	77*	13	2	74
4962 disk		303**			
-1	36120		60	2	273
-1F	36600		60	2	273
-2	36120		60	2	273
-2F	36600		60	2	273
-3	54180		60	3	182
-4	54180		60	3	182
4963 disk		360***			
-23	92160		64	4	128
-29	114560		64	5	102
-58	229632		64	10	51
-64	252032		64	11	46

* 73 cylinders are available for data (001-073) on type 1 diskettes. 74 cylinders are available for data (001-074) on type 2 diskettes. On both types, 2 cylinders are reserved for alternate tracks and 1 cylinder is reserved for IPL and volume identification.

** 301 cylinders are available for data (000, 002-301); cylinder 001 is reserved for alternate sector assignments; 302 is reserved for CE use.

*** 358 cylinders are available for data (0-357), while cylinder 358 is reserved for alternate sectors and cylinder 359 is reserved for CE use.

PART II - SYSTEM GENERATION AND CONFIGURATION

The creation of a customized supervisor is a two step process. Step 1 is a definition phase. Step 2 is the generation phase.

In step 1, you define the configuration of the system by preparing configuration statements which describe the attributes of the devices (such as disks, diskettes, and terminals) you want your system to support. You also define the number and size of the partitions that will be available in your system. Configuration statements are described in "Chapter 6. System Configuration" on page 75.

In step 2, you enter your configuration statements and assemble them. Then you modify the system-supplied INCLUDE file, $LNKCNTL, ensuring that all the support you require is built into the supervisor. The linkage editor combines the supervisor definition with the supervisor functions you selected to create a customized supervisor.

The volume label, tape ID, and the label of a terminal statement must all be uniquely defined. Otherwise, unpredictable results may occur.

No device (other than disk) can be defined more than once.

The system generation process is described in "Chapter 7. System Generation" on page 115.

MC - The Series/1 is the controlling station on a multipoint line. The adapter should be jumpered with DTR permanently enabled and multipoint line should not be jumpered.

MT - The Series/1 is a tributary station on a multipoint line. The adapter should be jumpered for multipoint tributary operation with DTR permanently enabled.

RETRIES= The number of attempts which should be made to recover from common error conditions before posting a permanent error.

MC= NO - The binary synchronous adapter located at the address specified in the ADDRESS= operand is either a medium speed, single line feature card or a high speed, single line feature card.

YES - The binary synchronous adapter located at the address specified in the ADDRESS= operand is part of a multi-line controller feature configuration. When generating supervisors using multi-line controller attachments, note the following:

- The character string YES must be specified. Any other character string will be equivalent to NO.
- All multi-line feature cards must start at a base address ending with either X'0' or X'8'. A BSCLINE statement must exist for the line at this base address if any of the other lines of the multi-line attachment are to be used.

END= YES, for the last BSCLINE statement in the system definition module.

Examples:

```
BSCLINE ADDRESS=28,TYPE=PT,RETRIES=10,MC=NO
BSCLINE ADDRESS=30,TYPE=SM,RETRIES=2,MC=YES,END=YES
```

DISK

DISK - Define Direct Access Storage

DISK defines the direct access storage devices and logical volumes to be supported in the generated system. All DISK statements must be grouped together. The last DISK statement must include an END=YES specification.

DISK is only needed in the system generation process. Refer to "Chapter 3. Data Management" on page 45 for a general discussion of direct access storage organization, functions, and naming conventions.

The disk Volser name must be unique for the system.

Syntax

```
blank      DISK     DEVICE=,ADDRESS=,VOLSER=,VOLORG=,
                    VOLSIZE=,VERIFY=,BASEVOL=,FHVOL=,
                    LIBORG=,END=,TASK=

Required:
    For 4964, 4966:  DEVICE=,ADDRESS=
    For 4962, 4963:  DEVICE=,ADDRESS=,VOLSER=,VOLSIZE=
    For 4962, 4963 (with fixed head): DEVICE=,ADDRESS=
                     VOLSER=,VOLSIZE,FHVOL=
Defaults: LIBORG=241 for 4962-1 or 4962-2 primary volume
          LIBORG=1 for secondary volume
          LIBORG=361 for 4962-1F or 4962-2F primary vol
          LIBORG=129 for 4963-64 or 4963-58 primary vol
          LIBORG=129 for 4963-29 or 4963-23 primary volum
          END=NO,TASK=NO,VERIFY=YES
```

Operands	Description
DEVICE=	4964, to define a 4964 Diskette Drive, or one of the following for the six models of the 4962 disk:

```
4962-1  for a 9.3  megabyte unit
4962-1F for a 9.3  megabyte unit
                   with fixed heads
4962-2  for a 9.3  megabyte unit
                   with a diskette unit
4962-2F for a 9.3  megabyte unit
                   with fixed heads
                   and a diskette unit
4962-3  for a 13.9 megabyte unit
4962-4  for a 13.9 megabyte unit
                   with a diskette unit
```

or

one of the following for the four models of the 4963 disk:

```
4963-29 for a 29 megabyte unit
4963-23 for a 23 megabyte unit with fixed heads
4963-64 for a 64 megabyte unit
4963-58 for a 58 megabyte unit with fixed heads
```

or

4966, to define a 4966 Diskette Magazine Unit.

Note: If 4962 or 4963 is specified, VOLSER= must be specified; LIBORG= may be specified.

ADDRESS= The hexadecimal address of the unit. This parameter is required for primary volumes only.

IPL devices must be at the following addresses:

Device	Address
4962	X'03'
4963	X'48'
4964	X'02'
4966	X'22' (IPL must occur from slot 1)

VOLSER= Volume label (1-6 characters) to be assigned to the unit. This operand is required if the DEVICE=4962- or DEVICE=4963- is specified. Otherwise, it is ignored.

VOLORG= The physical cylinder number of the first cylinder of the volume. Cylinder numbering begins with zero. A primary volume must begin at cylinder zero. (Refer to Figure 9 on page 58.)

DISK

VOLSIZE= The size of the volume in physical cylinders. The minimum value allowed is 2. (Refer to Figure 9 on page 58.)

VERIFY= NO, to omit the WRITE-with-verify option. YES, to cause each WRITE to be verified. YES is the default. This parameter is required for primary volumes only.

Note: You should choose the VERIFY=YES option for volumes containing critical data. This causes a slight performance degradation but improves reliability. With the YES option, each WRITE is immediately followed by a READ, thus lengthening the operation by the time it takes the unit to make one revolution.

BASEVOL= The volume label of the primary volume if a secondary volume is being defined.

FHVOL= The volume label to be assigned to the automatically generated secondary volume if the DISK statement is defining a primary volume on any 4962 or 4963 having fixed heads.

LIBORG= The origin, by number of records, of the directory on the volume. Defaults are described under 'Syntax'. This operand is only applicable when DEVICE=4962 or 4963 and is intended for special use when the initial portion of the volume is reserved for other storage.

END= YES, for the last DISK statement in the system definition module.

TASK= YES, to cause a new I/O task to be generated. This task will be used to service I/O requests for this and subsequent primary volumes until a new DISK statement with TASK=YES is encountered. NO, or omit, if a new task is not required. This operand is valid only for primary volumes and is optional.

Specifying TASK=YES on a primary volume allocates a Task Control Block that is used in servicing READ and WRITE requests for the group of devices being defined. The effect is to allow READ and WRITE requests to proceed in parallel with requests to other groups of devices. The resulting overlap may significantly improve performance when concurrent requests to different groups of devices occur. To achieve maximum flexibility and performance, you should specify TASK=YES on each primary volume. Additional storage required for each TASK=YES is 128 bytes.

- Processor time requirements

These items vary with each installation.

PARTS= The number of 2K (1K=1024 bytes) blocks of storage to be assigned to each partition. Use only if STORAGE= is specified as greater than 64. Enter a list showing the maximum size of each partition. Up to eight partitions can be defined for the 4955, up to two for the 4952, and one for the 4953. The list must contain the same number of entries as the list coded for MAXPROG=.

The method for calculating the maximum size for partition one is as follows:

Determine the available storage in the first 64K by subtracting the size of the supervisor from 64K. See Appendix A to estimate the supervisor size.

The size of partition one is determined when you IPL, by using the smaller of:

- The size you define in the PARTS= parameter
- 64K minus the size of the supervisor

The maximum value that can be specified is 32; the minimum is 2. When specifying the size to be assigned to partition one, you may code 32 rather than calculating the value, if you wish partition one to have all storage not used by the supervisor. Otherwise, you must calculate the size of partition one.

The Multiple Terminal Manager partition size can be calculated by using the information in the Communications and Terminal Applications Guide.

DATEFMT= The format to be used when the date is displayed (PRINDATE or $W) or when entering the date via $T. A return code is set in response to a GETTIME request with the DATE option.

Specify MMDDYY for a date format of month.day.year. Specify DDMMYY for a date format of day.month.year. MMDDYY is the default.

Note: Timer support must be included in your supervisor in order to have date support.

SYSTEM

IABUF= The maximum number of interrupts that may be buffered by the task supervisor. The default value is adequate for most systems. The value should be increased if the system could be overloaded by a large number of interrupts. (The system will stop or enter a continuous run loop.) Each increment increases the supervisor storage requirements by eight bytes.

COMMON= The label of the last supervisor address to be mapped in every partition. The value will be automatically rounded upward to a 2K byte boundary. To map the entire supervisor, specify COMMON=START. To map only the supervisor data areas, specify COMMON=EDXSVCX. The default, COMMON=EDXSYS, implies no mapping. Refer to "$SYSCOM - Define Optional Common Data Area" on page 113 for additional information.

Example 1

```
SYSTEM   STORAGE=96,MAXPROG=(3,2,3),          C
      PARTS=(32,6,10)
```

This three partition system is possible on a 96KB 4955 and maps as follows:

PARTITION 1	28KB SUPERVISOR	36KB USER SPACE
PARTITION 2	12KB USER SPACE	
PARTITION 3	20KB USER SPACE	

1. Partition 1 is 36KB and can execute up to three programs concurrently.
2. Partition 2 is 12KB and can execute up to two programs concurrently.
3. Partition 3 is 20KB and can execute up to three programs concurrently.

Note: The 28KB supervisor size is used for illustrative purposes only.

Example 2

PARTITION 1	28KB SUPERVISOR	32KB USER SPACE
PARTITION 2	36KB USER SPACE	

1. Because COMMON=START was specified, the supervisor is mapped in both partition 1 and partition 2, providing direct addressability to the supervisor for all programs that execute on this system.

2. Partition 1 is 32KB and can execute up to three programs concurrently.

3. Partition 2 is 36KB and can execute up to four programs concurrently.

Note: The 28KB number for the supervisor is used for illustrative purposes only.

TAPE

TAPE - Define Tape Device (Version 2 only)

TAPE defines the tape devices on a system. One TAPE statement is required for each tape device on the system. It is recommended that you group all DISK statements together, followed by all the TAPE statements. The last TAPE or DISK statement must include an END=YES specification. The tape ID must be a unique name.

Syntax

```
blank  TAPE DEVICE=,ADDRESS=,DENSITY=,LABEL=,ID=,
            TASK=,END=
    Required: DEVICE=,ADDRESS=,ID=
    Defaults: DENSITY=1600,LABEL=SL,TASK=NO,END=NO
```

Operands	Description
DEVICE=	Device type (4969 to define IBM 4969 tape unit)
ADDRESS=	A two digit hexadecimal number specifying the address assigned to the unit
DENSITY=	Tape density to be used for this device (800,1600,DUAL). When DUAL is coded, density defaults to 1600 BPI.
LABEL=	Type of processing to be done on this device. Standard label (SL), non-label (NL), and bypass label processing (BLP) are the only types supported.
ID=	A one-to six-character name that is associated with the device. This operand is used primarily for specifying the drive when NL or BLP is used.
TASK=	YES, causes a new I/O task to be generated. This task is used to service I/O request for this and subsequent tapes until a new TAPE statement with TASK=YES is encountered. For best performance, specify TASK=YES for each tape unit that has a controller.
END=	YES, for the last statement in the DISK/TAPE sequence.

Example

```
TAPE  DEVICE=4969,ADDRESS=4C,DENSITY=1600,          X
      LABEL=SL,ID=$TAPE1,                           X
      TASK=YES,END=YES
```

Note: END=YES is specified only once for the DISK/TAPE definition statements.

TERMINAL

TERMINAL - Define Input/Output Terminals

TERMINAL defines each input/output terminal to be supported in the generated system. Output only devices, such as line printers, are also specified with TERMINAL statements. All TERMINAL statements must be grouped together with the last statement including an END=YES specification.

A TERMINAL statement specifying DEVICE=VIRT can be entered in an application program provided exactly the same statement is entered in the system configuration program. All TERMINAL statements within the application program are automatically converted to an IOCB statement. The label on the TERMINAL statement is used for the label and the operand of the IOCB statement. Labels on all terminal statements must be unique for the system.

Before preparing your TERMINAL statements, you need to know the characteristics of your terminals, the way they will be attached to your Series/1, and how you plan to use them in your application. Review the appropriate hardware manuals, the topic entitled "Terminal I/O" in the Language Reference, and the appropriate topics in Communications and Terminal Applications Guide.

If you use the Remote Management Utility and need the PASSTHRU function, two virtual terminals are required. For a detailed description of the PASSTHRU function see the Remote Management Utility chapter in Communications and Terminal Applications Guide. See Figure 10 on page 107 for a sample configuration.

Syntax

```
label TERMINAL DEVICE=,ADDRESS=,PAGSIZE=,LINSIZE=,
               CODTYPE=,TOPM=,BOTM=,NHIST=,LEFTM=,RIGHTM=
               OVFLINE=,LINEDEL=,CHARDEL=,CRDELAY=,ECHO=,
               BITRATE=,RANGE=,LMODE=,ADAPTER=,COD=,CR=,
               LF=,HDCOPY=,ATTN=,PF1=,SYNC=,SCREEN=,PART=
               DI=,DO=,PI=,END=,TYPE=

Required: DEVICE= ,and one of the following:

•   ADDRESS= except for DI/DO terminals

•   DI=,DO=,PI= for DI/DO terminals

Defaults: PART=1,END=NO
```

Operands	Description
DEVICE=	One of the following codes for the indicated device:
TTY	A 3101 Display Terminal or other ASCII Terminal attached via Teletypewriter Adapter (7850)
4979	4979 display station attached via 3585 Adapter
4978	4978 display station attached via RPQ D02038
4974	4974 matrix printer attached via 5620 Adapter
4973	4973 line printer attached via 5630 Adapter
2741	2741 communications terminal attached via 1610 controller
4013	Graphics terminal attached via 1560 adapter (Refer to Communications and Terminal Applications Guide for hardware considerations.)

TERMINAL

ACCA A 3101 Display Terminal or other ASCII terminal attached via 1610 controller or 2091 controller with 2092 adapter or 2095 controller with 2096 adapter (Refer to Communications and Terminal Applications Guide for hardware considerations.)

PROC Processor-to-processor communication

VIRT Inter-program communication. (Refer to "Chapter 14. Inter-Program Communications" on page 279.)

ADDRESS= The address (in hexadecimal) of the device. (Refer to "Chapter 14. Inter-Program Communications" on page 279 for the use of this parameter in connection with virtual terminal communications.)

PAGSIZE= The physical page size (form length) of the I/O medium. Specify a decimal number between 1 and the maximum value which is meaningful for the device. For printers, specify the number of lines per page, or for screen devices the size of the screen in lines. This operand is not required for the 4978/4979 display; its value is forced to 24. For a printer the default is 66.

CODTYPE= The transmission code used by the terminal. Specify either ASCII, EBCDIC, EBCD (PTTC/EBCD), CRSP (PTTC/correspondence), or EBASC (8 bit data interchange code) as in the following table:

	Adapter			
	7850	1610	2091/2092	2095/2096
DEVICE=TTY	ASCII (default)	N/A	N/A	N/A
DEVICE=2741	N/A	EBCD or CRSP	N/A	N/A
DEVICE=ACCA	N/A	EBASC (default)	EBASC (default)	ASCII

LINSIZE= The maximum length of an input or output line for the device. The maximum line length cannot exceed 254 characters. The value of this operand can be less than the maximum which the device can accommo-

date (for example, 80 for the 4978/4979 display station or 132 for the 4974 printer), but the value is then fixed and cannot be altered dynamically. For a printer the default is 132.

TOPM= The top margin (a decimal number between zero and PAGSIZE-1) to indicate the top of the logical page within the physical page for the device.

NHIST= The number of history lines to be retained when a page eject is performed on the 4978/4979 display. The line at TOPM+NHIST corresponds to logical line zero for purposes of the terminal I/O instructions. When a page eject (LINE=0) is performed, the screen area from TOPM to TOPM+NHIST-1 will contain lines from the previous page. This operand is meaningful for roll screens only. (See the discussion of the SCREEN operand which follows.)

BOTM= The bottom margin, the last usable line on a page. Its value must be between TOPM+NHIST and PAGSIZE-1. If an output instruction would cause the line number to increase beyond this value, then a page eject, or wrap to line zero, is performed before the operation is continued.

LEFTM= The left margin, the character position at which input or output will begin. Specify a decimal value between zero and LINSIZE-1.

RIGHTM= A value (between LEFTM and LINSIZE-1) that determines the last usable character position within a line. Position numbering begins at zero.

OVFLINE= YES, if output lines that exceed the right margin are to be continued on the next line. This condition arises when the system buffer or user buffer, if provided) becomes full and you have taken no specific action in your application program (such as forms control commands) to write the buffer to the device.

LINEDEL= A two-digit hexadecimal character that defines the character the operator will enter when he wishes to restart an input line. In some cases, input of this character causes a repeat of the previous output message. Usually, this operand is not meaningful for devices such as the 4979 display station, whose input is formatted locally before entry. (For the ACCA terminals attached via the 1610 or 2091 controllers and the 2092 adapter, code in mirror image. Refer below for a description of mirror images.)

CHARDEL= A two-digit hexadecimal character which indicates deletion of the previous input character. It is meaningful only for devices whose mode of transmission is one character at a time, as described in the LINEDEL operand. For the ACCA terminals attached via the 1610 or 2091 controllers and the 2092 adapter, enter in mirror image.

CRDELAY= The number of idle times required for a carriage return to complete for teletypewriter devices. If printing occurs during the carriage return, CRDELAY is too small. For interprocessor communications (DEVICE=PROC), refer to the Communications and Terminal Applications Guide.

ECHO= NO, for devices that do not require input characters to be written back (echoed) by the processor for printing.

YES (the default) is appropriate for most devices connected through the teletypewriter adapter. NO is required for ACCA. See the LF parameter description regarding suppression of the echo of the CR character.

BITRATE= The rate (in bits per second) that this terminal will be operating. (Used with ACCA, 2741 and PROC support only.)

RANGE= Enter HIGH or LOW to match hardware jumper that is installed on the adapter card. (Used with ACCA, 2741 and PROC support only.)

LMODE= SWITCHED or PTTOPT. If this line is used with a switched connection, then enter SWITCHED. Otherwise, enter PTTOPT. (Used with ACCA support only.)

ADAPTER= One of the following to indicate the ACCA type:

SINGLE For the single line controller

TWO For the eight line controller with up to two lines active

FOUR For the eight line controller with up to four lines active

SIX For the eight line controller with up to six lines active

EIGHT For the eight line controller with up to eight lines active

All multiple line feature cards must start at a base address ending with with X'0' or X'8'. A terminal statement with DEVICE=ACCA must exist for the line at the base address. Furthermore, the terminal defined as the base address must be specified as the first terminal for the multiline controller. The remaining terminals defined on the multiline controller (if any) must immediately follow the base address terminal and should be in ascending order by address.

Note: For DEVICE=2741, only SINGLE is allowed.

This should match the jumpers on the controller cards. (Refer to the Communications and Terminal Applications Guide for hardware considerations.)

COD= Additional characters, other than the CR=, ATTN=, and LINEDEL= values, that will terminate a READ operation. (COD means change of direction, for example, READ to WRITE.) (Used with ACCA only.) Code in mirror image as follows:

```
COD=11
or
COD=(12,B6,42...)
```

From one to four COD characters may be entered.

CR= The single character to be tested to determine if a new line function is to be performed. (Code in mirror image for ACCA terminals attached via 1610 or 2091 controllers with a 2092 adapter.)

LF= The character to be sent to the terminal when a new line function is to be performed. Code in mirror image for ACCA terminals attached via the 1610 or 2091 controllers with the 2092 adapter. If the same value is coded for LF= as was coded (or defaulted) for CR= then the CR character which terminates an input operation will not be echoed to the terminal; the terminal is assumed to be an auto-line feed device.

HDCOPY= Support for the 4978/4979 display station includes a means of printing the contents of the display screen on a hardcopy device for permanent record. (For an explanation of the hardcopy feature, refer to Utilities, Operator Commands, Program Preparation, Messages and Codes). The hardcopy function is defined by coding:

HDCOPY=(terminal name, key),

terminal name — The symbolic name of the terminal to which the hardcopy contents will be directed

key — The code of the program function key which is to invoke the function. For example, HDCOPY=($SYSPRTR,4) designates $SYSPRTR as the hardcopy device and PF4 as the activating key. If the hardcopy terminal name alone is specified, as for example in HDCOPY=$SYSPRTR, then the default is PF6. Note: The terminal specified (Terminal name) must not be defined with ATTN=NO.

ATTN= NO, if the attention key and the 4978/4979 PF keys are to be disabled for the terminal. Such disabling is then permanent for the generated system. If you do not specify ATTN=, the default is the ATTN key.

LOCAL, to limit the attention functions to those defined by ATTNLISTs within programs loaded from the terminal.

NOSYS, to exclude only the system functions ($L, $C, etc.).

NOGLOB, to exclude only the global ATTNLIST functions. (GLOBAL is the ATTNLIST of all programs in the same partition at one time.)

Note: This operand can also be entered with a two-digit hexadecimal character for the attention key if the system default is not desired.

The attention key can be redefined with a two-digit hexadecimal character for the 4978/4979 displays or ASCII terminals.

For terminals attached via the 1610 or 2091 controllers and the 2092 adapter, use mirror image. (Refer to "Mirror Image" on page 109 for a discussion of mirror image.)

For the 3101 display terminal, enter X'D9' if the terminal is attached via the 1610 or 2091 controllers and X'9B' if it is attached via the 2095 controller. You may have the Mark Parity Switch set on (refer to the IBM 3101 Display Terminal Description

GA18-2034, for information on switch settings).

The default for ATTN for ASCII terminals is ASCII X'1B', the ESC key. The mirror image of X'1B' is X'D8'. Note: If the terminal being defined is specified in the HDCOPY= parameter of an other terminal, do not code ATTN=NO.

Note: If the terminal being defined is a teletypewriter device to be used as $SYSPRTR, do not code ATTN=NO.

PF1= For the 4978 display, code the two-digit hexadecimal character which is to be interpreted as Program Function key 1. Successive values are then interpreted as PF2 and PF3.

The default for this operand is 2.

SYNC= This keyword applies to virtual terminal communications. Code SYNC=YES if synchronization events will be posted to this virtual terminal.

This means that attempted actions over the virtual channel will be indicated in the task control word. This allows the two terminals to synchronize their actions so that when one terminal is writing, the other is reading.

SYNC=NO is the default.

SCREEN= One of the following to indicate whether the terminal is a hardcopy or screen device:

YES or ROLL for screens which are to be operated like a typewriter.

For screen devices which are attached through the teletypewriter adapter, this indicates that a pause will be performed when a screen-full condition occurs during continuous output.

NO for hardcopy devices. For 4978 or 4979 devices, NO results in inhibiting the pause when the screen fills up (the screen acts as a roll screen).

STATIC for a full-screen mode of operation, if this mode is supported for the device.

Note: The initial terminal configuration should be STATIC only if the terminal is reserved for data display and data entry operations. Normal system operations, such as those directed to $SYSLOG or those involving the utility programs, assume a roll screen configuration. The application program can define the static screen configuration by means of the ENQT and IOCB instructions described in the Language Reference.

TERMINAL

PART= A number (1-8) to indicate the partition with which the terminal is normally associated.

This is valid only if the STORAGE= operand of the SYSTEM statement was specified to be greater than 64. You can change the partition assignment at execution time with the $CP Command described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

END= YES, for the last TERMINAL statement in a system definition module.

TYPE= Specify DSECT to generate a CCB DSECT in your program. for programs processed by $S1ASM. Do not specify DSECT in programs processed by $EDXASM; use COPY CCBEQU elsewhere in your program.

The following three operands are for terminals connected via digital I/O only:

Operands	Description
DI=(address,termaddr)	
address	The digital input group address.
termaddr	The hardware subaddress (0-7) of the terminal defining the value used to select the terminal for digital input.
DO=(address,termaddr)	
address	The digital output group address
termaddr	The hardware subaddress (0-7) to define the digital output subaddress of the terminal
PI=(address,bit)	
address	The process interrupt group address.
bit	The bit (0-15) to define the particular interrupting point assigned to the terminal.

4013[4] (DI/DO Parallel Interface) TERMINAL Statement

```
TERMINAL  DEVICE=4013,DI=(80,01),DO=(87,01),                    C
          PI=(84,04),PAGSIZE=35,LINSIZE=72,                     C
          CODTYPE=ASCII,TOPM=0,BOTM=34,LEFTM=0,                 C
          RIGHTM=71,SCREEN=NO,OVFLINE=NO,                       C
          LINEDEL=7F,CHARDEL=08,CRDELAY=0,ECHO=YES,             C
          CR=0D,LF=0A
```

Remote Management Utility using the PASSTHRU function - TERMINAL Statements

```
CDRVTA    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,                 C
                    SYNC=YES,LINSIZE=132

CDRVTB    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA,                 C
                    SYNC=NO,LINSIZE=132
```

Note: This example shows a line size of 132. The maximum line size value is 254.
The names CDRVTA and CDRVTB are required.

The following statements are coded with values that are not defaults for parameters PAGSIZE, ATTN, CR, CHARDEL, LINEDEL, ADAPTER, BOTM, SCREEN, BITRATE, RANGE, and MODE. Use these values if the IBM 3101 Display Terminal is attached to your system. For DEVICE=ACCA, you must set the mark parity switch on (refer to the IBM 3101 Display Terminal Description, GA18-2033, for information on switch settings).

[4] Registered trademark of the Tektronix Corporation.

TERMINAL

IBM 3101 TERMINAL Statement (via 7850 adapter)

```
TERMINAL  DEVICE=TTY,ADDRESS=00,CRDELAY=4,                    C
          PAGSIZE=24,SCREEN=YES
```

IBM 3101 TERMINAL Statement (via 2095 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=60,BITRATE=110,                 C
          PAGSIZE=24,LINSIZE=80,                              C
          CODTYPE=ASCII,TOPM=0,BOTM=23,LEFTM=0,               C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,                    C
          LINEDEL=FF,CHARDEL=88,CRDELAY=0,ECHO=NO,            C
          RANGE=LOW,LMODE=PTTOPT,                             C
          CR=8D,LF=0A,ATTN=9B,ADAPTER=FOUR
```

IBM 3101 TERMINAL Statement
(via 1610 or 2091 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=6B,BITRATE=110,                 C
          PAGSIZE=24,LINSIZE=80,                              C
          CODTYPE=EBASC,TOPM=0,BOTM=23,LEFTM=0,               C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,                    C
          LINEDEL=FF,CHARDEL=11,CRDELAY=0,ECHO=NO,            C
          RANGE=LOW,LMODE=SWITCHED,                           C
          CR=B1,LF=50,ATTN=D9,ADAPTER=EIGHT
```

IBM 3101 Model 2 (block mode) under Multiple Terminal Manager TERMINAL Statement (via 1610 or 2091 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=08,BITRATE=2400,          C
          PAGSIZE=24,LINSIZE=80,                        C
          CODTYPE=EBASC,TOPM=0,BOTM=23,LEFTM=0,         C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,              C
          LINEDEL=FF,CHARDEL=11,CRDELAY=0,ECHO=NO,      C
          RANGE=HIGH,LMODE=PTTOPT,                      C
          CR=B1,LF=50,ATTN=A8,ADAPTER=SINGLE
```

IBM 3101 Model 2 (block mode) under Multiple Terminal Manager TERMINAL Statement (via 2095 controller)

```
TERMINAL  DEVICE=ACCA,ADDRESS=61,BITRATE=2400,          C
          PAGSIZE=24,LINSIZE=80,                        C
          CODTYPE=ASCII,TOPM=0,BOTM=23,LEFTM=0,         C
          RIGHTM=79,SCREEN=YES,OVFLINE=NO,              C
          LINEDEL=FF,CHARDEL=88,CRDELAY=0,ECHO=NO,      C
          RANGE=HIGH,LMODE=PTTOPT,                      C
          CR=8D,LF=0A,ATTN=15,ADAPTER=FOUR
```

Mirror Image

Mirror image is used by ASCII terminals attached via the 1610 or 2091 controllers and the 2092 adapter. Mirror image reverses the bit pattern for data. For example, the EBCDIC character 1 would look as follows:

X'F1' EBCDIC

X'31' ASCII

X'8F' Mirror Image EBCDIC

X'8C' Mirror Image ASCII

When using XLATE=NO on Event Driven language instructions PRINTEXT and READTEXT, the data sent must be in mirror image. Data received is in mirror image.

ASCII Terminal Codes

Terminals and other devices equivalent to the Teletype ASR 33/35 are referred to in this document as "ASCII terminals." These terminals may be attached to the Series/1 in a variety of ways. Note that while the bit representation of a character appearing at the terminal is the same for all the attachments, two different representations for a given character are used internally.

One representation is ASCII, in which the characters appear in main storage in ASCII code. This code is used by features #7850, #2095, and #2096.

The other representation is the Eight Bit Data Interchange Code. It is used by the 1610 and 2091 controllers and the 2092 adapter. This representation is the mirror image within a **byte** of the ASCII representation. The bits appear swapped end-for-end within each byte.

Note also that ASCII terminals may use even, odd, or no parity. The parity bit appears as the high order bit in ASCII code and as the low order bit in Eight Bit Data Interchange Code. You must incorporate the proper parity, if any, within the data characters. You must also incorporate the proper parity, if any, within the control characters specified by the LINEDEL, CHARDEL, COD, CR, and LR parameters of the TERMINAL statement.

Symbolic Reference to Terminals

The optional label on the TERMINAL statement is used to assign a name to the device for purposes of reference by the application program. Three such names have special meaning to the supervisor and should be assigned to the appropriate device:

$SYSLOG Names the system logging device or operator station, and must be defined in every system. In the starter supervisor, $SYSLOG defines a 4978 display station.

CHAPTER 7. SYSTEM GENERATION

To generate an Event Driven Executive system, you must have access to a Series/1 capable of preparing the supervisor program and application programs. System generation requires that the following licensed programs be installed:

- Basic Supervisor and Emulator
- Event Driven Executive Utilities
- Event Driven Executive Program Preparation Facility (requires 26K bytes of storage)

  or

  Series/1 Macro Assembler and Macro Library (requires 16K bytes of storage. This will allow system generation in a Series/1 program preparation configuration which includes a 4955 processor with a minimum of 48K bytes of storage.)

The Program Preparation Facility enables you to prepare programs to be executed on any Series/1 that has the required hardware configuration and licenses.

GENERATING THE SUPERVISOR

Creating a supervisor program tailored to your Series/1 hardware configuration requires the use of several of the utilities and program preparation programs; these include:

- Disk data set management ($DISKUT1)
- Text editor ($EDIT1N)

  or

  Full-screen editor ($FSEDIT)
- Batch job stream processor ($JOBUTIL)
- Event Driven Language compiler ($EDXASM)

  or

  Series/1 Macro Assembler ($1ASM) and Macro Library

- Linkage editor ($LINK)
- Object module conversion ($UPDATE)

You should become familiar with these utilities, especially the text editors, before attempting to generate the supervisor. These utilities are described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

The following major steps are required:

- Step A. Allocate required data sets.
- Step B. Edit $EDXDEF, the system configuration file, to match your hardware configuration..
- Step C. Edit $LNKCNTL, the system-supplied INCLUDE file, to specify which supervisor program object modules are to be included in your supervisor.
- Step D. Edit $SUPPREP, the system-supplied job stream processor file, to use your allocated data sets.
- Step E. Use $JOBUTIL and the procedure file created in Step D to:
  - Assemble the supervisor definition module created in Step B
  - Link edit the resulting object module with the other necessary supervisor object modules using the link edit control data set created in Step C.
  - Using $UPDATE, convert the output of the link edit process into an executable supervisor, and store it in a data set named $EDXNUCT.
- Step F. Test the created supervisor on a disk based system.
- Step G. Verify the system generation process (optional).

Step A - Allocate Required Data Sets

1. IPL the system from disk volume EDX002.
2. Load utility program $DISKUT1 and use the AL command to allocate the following data sets on volume EDX002. All data sets must be specified as TYPE=DATA.

Data Set Name	Number of Records
EDITWORK	200
ASMOBJ	250
ASMWORK	250
SUPVLINK	450
LEWORK1	400
LEWORK2	150

Note: The actual size of the data set depends on the size of the supervisor being generated.

If you plan to use the utility $EDITIN to edit $EDXDEF and $LNKCTRL, you must allocate data sets $EDXDEFS (35 records) and $LNKCTRL (50 records) on EDX002.

Step B - Edit $EDXDEF to Match Hardware Configuration

Edit $EDXDEF to match your hardware configuration:

1. Load utility program $EDIT1N or $FSEDIT and specify EDITWORK as the reply to WORKFILE=.

2. Read the supplied data set $EDXDEF from volume ASMLIB. Figure 11 on page 133 shows a sample configuration of $EDXDEF. The supplied configuration can be seen in the Program Directory.

   The first time you use EDITWORK as a work file for the text editor, you will be asked if you can use the EDITWORK data set as a work data set; respond YES and continue.

3. Add to or delete from the contents of EDITWORK as necessary to create a set of system configuration statements. (System configuration statements are described in "Chapter 6. System Configuration" on page 75.) Some printer on the Series/1 should be designated as $SYSPRTR. When editing ensure that:

   - Continuation indicators in column 72 are not removed.
   - If required, a continuation character is placed in column 72 and the statement is continued in column 16 of the next line
   - A field does not extend beyond column 71

   The editing process consists of the following procedure:

   a. Calculate the total amount of storage available, the number of partitions desired, and the number of 2K blocks of storage desired for each partition. This information is inserted into the SYSTEM statement to define the characteristics of the processor. Refer to "Chapter 6. System Configuration" on page 75 for a description of the SYSTEM statement.

   b. Define the hardware features to be supported, using the appropriate system configuration statements (TIMER, SENSORIO, HOSTCOMM, BSCLINE, EXIODEV, DISK, TERMINAL, TAPE).

   c. Define the direct access storage devices and logical volumes to be supported in the generated system, using the DISK system configuration statement. Sample DISK configuration statements are supplied for each device in the $EDXDEF data set on ASMLIB. Refer to "Chapter 3. Data Management" on page 45 for storage capacities of the supported direct access storage devices. With this information, you can define your disk volumes.

The only restrictions are (1) that you define the required Event Driven Executive volumes (EDX002, EDX003, ASMLIB) in addition to your volumes and (2) that you follow the rules pertaining to library origins and maximum volume sizes.

Note: Optional software products may require additional volumes. Volume requirements are supplied with the product documentation.

d. Define the characteristics of all printers, displays, and teletypewriters, using the TERMINAL statement. Examples of various types of TERMINAL statements are included in the $EDXDEF data set.

Note: Check the speed of your 3101 in the terminal statement. The speed must match the 3101 switch settings.

4. Save the final version of the definition statements in the data set $EDXDEFS on volume EDX002.

Step C - Specify Object Modules

Edit $LNKCNTL to specify which supervisor program object modules are to be included.

1. Read data set $LNKCNTL from volume ASMLIB. The supplied contents of $LNKCNTL are shown in the following tables; footnotes are provided on required usage. The $LNKCNTL data set supplied with Version 1 does not include TAPE support.

```
*************************************************************
* SYSTEM SUPPORT -- INITIALIZATION
*************************************************************
 INCLUDE EDXINIT,XS2002  *H*   SUPERVISOR INITIALIZATION
 INCLUDE DISKINIT,XS2002 *M*   DISK(ETTE) INITIALIZATION
*INCLUDE TAPEINIT,XS2002 *M*   TAPE INITIALIZATION
 INCLUDE LOADINIT,XS2002 *C*   PROGRAM LOADER INITIALIZATION
 INCLUDE RW4963ID,XS2002 *M*   4963 FIXED HEAD REFRESH SUPPORT
 INCLUDE TERMINIT,XS2002 *1*   TERMINAL INITIALIZATION
 INCLUDE INIT4978,XS2002 *M*   4978 DISPLAY INITIALIZATION
*INCLUDE INIT4013,XS2002 *M*   DIGITAL I/O TERMINAL INIT
*INCLUDE $ACCARAM,XS2002 *3*   ACCA MULTI-LINE ADAPTER RAM LOAD
*INCLUDE BSCINIT,XS2002  *7*   BISYNC (BSCAM) INITIALIZATION
*INCLUDE $BSCARAM,XS2002 *7*   BISYNC MULT-LINE ADAPTER RAM LOAD
*INCLUDE TPINIT,XS2002   *8*   HCF (TPCOM) INITIALIZATION
*INCLUDE TIMRINIT,XS2002 *6*   4953/4955 TIMER INITIALIZATION
*INCLUDE CLOKINIT,XS2002 *6*   4952 TIMER INITIALIZATION
*INCLUDE SBIOINIT,XS2002 *M*   SENSOR I/O INITIALIZATION
*INCLUDE EXIOINIT,XS2002 *M*   EXIO INITIALIZATION
```

NOTES

```
*0*   Must be included first and in this order
*1*   Required if any terminals are installed, including 4973
*     or 4974
*2*   Required if IOSTTY, IOS2741, or IOSACCA is included
*3*   Required if non-2741 terminals are on ACCA
*4*   Required if IOSTTY is included
*5*   Either TREBCD or TRCRSP or both are required if IOS2741
*     is included, depending on the code used by the 2741
*     terminals - correspondence or ASCII
*6*   Attached TIMERS (feature 7840) and the 4952 native TIMER
*     are mutually exclusive.  Select the TIMER support
*     required for your configuration or none if no TIMER
*     support is required.
*7*   Required for binary synchronous communication using
*     BSCREAD/BSCWRITE or Remote Management Utility support.
*8*   Required for communication to a S/370 with the EDX Host
*     Communication Facility
*9*   Required if any Sensor I/O support is to be used
*     (AI,AO,DI,DO, or PI)
*A*   One, but not both, of these modules is required
*B*   Required if the in storage program check/machine check
*     log is to be kept
*C*   Required if programs are to be loaded from disk(ette).
*     If not included, an application program must be link
*     edited with the supervisor.
*D*   One, but not both, of these modules is required
*E*   Required for data formatting operations (GETEDIT,
*     PUTEDIT, FORMAT)
*F*   Required for queueing operations (FIRSTQ, NEXTQ, LASTQ,
*     DEFINEQ)
*G*   Required for program debugging ($DEBUG)
```

```
*H*   Required and must follow all of the previously listed
*     modules.
*     All other initialization modules must follow EDXINIT.
*J*   For starter supervisor use only
*K*   There are two versions of this module.  This one is
*     for systems that support the address translator
*     feature of the 4952 and 4955 processors.  Include this
*     version if your system is to support both the function
*     the module implements and the address translator
*     feature. (XL)
*L*   There are two versions of this module.  This one is
*     for systems that do not support the address translator
*     feature of the 4952 and 4955 processors.  Include this
*     version if your system is to support the function
*     the module implements, but not the address translator
*     feature. (UN-XL)
*M*   Optional module; required if device or feature is to be
*     supported.
*N*   Required if using Remote Management Utility with PASSTHRU
*     function.
  END
```

Note: You should include DDBFIX and CCBFIX with the other system intialization modules if you wixh to regenerate the starter system.

2. Enter an asterisk (*) in column one (1) of each INCLUDE statement not required to create your supervisor. The asterisk makes the statement a comment and the module with the asterisk is not included in your supervisor. Be sure that the system definition statements created in Step B agree with the modules you include in this step.

   The modules with note L can be used if your generated system is to execute either on a Series/1 without the address translator feature or on a 64KB 4952 processor. These modules do not support the address translator. The SYSTEM configuration statement must specify STORAGE as 64 or less and PARTS may not be specified.

3. Save the edited version of $LNKCNTL in a data set named LINKCNTL on EDX002.

Step D - Assemble and Link Edit the Supervisor

The $SUPPREP procedure below specifies the use of the $EDXASM compiler. If the $S1ASM assembler is required, the appropriate procedure statement changes must be made.

Edit $SUPPREP to use your allocated data sets.

1. Read the data set $SUPPREP from volume ASMLIB. Figure 10 on page 125 shows $SUPPREP.

3. Test the supervisor by executing utility programs that exercise the various supervisor components (such as disk I/O, sensor I/O, etc.)

Notes:

- If the new supervisor fails to operate correctly, you must restore the original contents of $EDXNUC by IPLing from a diskette. Use $COPY or $COPYUT1 to copy the starter supervisor from diskette UT3001 orUT4001 to $EDXNUC on EDX002.
- If any errors are encountered, repeat steps B through E of this procedure.
- If you relocated any volumes in a tailored system generation (particularly EDX002), copy the new supervisor into the $EDXNUC data set on a copy of the utility diskette (UT3001 orUT4001) and perform a complete system installation.
- The actual addresses of CSECT and ENTRY point labels in the $EDXNUCT or $EDXNUC modules stored on disk will be X'100' greater than those shown on the link edit map. This is because $UPDATE adds a 256 byte header to all $EDXNUCx modules.
- If you have a 4966 Diskette Magazine Unit, the door must be closed during IPL.

Step G - Verify the System Generation Process

To verify that the system generation has been performed successfully:

1. Assemble and execute the sample program CALCSRC.

   Note: CALCDEMO source instructions are located in the data set CALCSRC on the disk volume EDX002. To assemble CALCDEMO, refer to the procedure for program preparation described in Utilities, Operator Commands, Program Preparation, Messages and Codes.

2. When the assembly is complete, load the test program into storage for execution by using the $L operator command.

3. When you receive the prompts A= and B=, enter any decimal integer values less than 2 billion, followed by a carriage return or ENTER after each entry.

   A sample of the entries and resulting output follows:

```
> $L CALCDEMO

CALCDEMO      3P,10:59:55, LP= 7F00
Press ATTENTION and enter CALC or STOP
> CALC

A = 12
B = 52

A + B =            64
A - B =           -40
A * B =           624
A / B =             0 REMAINDER =           12
Press ATTENTION and enter CALC or STOP
> CALC
```

OTHER CONSIDERATIONS

Control and Image Stores for the 4978

The system includes modules $4978IS0, $4978CS0, $4978CS1, and $4978IS1. These four modules are the 4978 stores for the D02056 keyboard.

If $4978IS0 and $4978CS0 are present in the IPL volume, the image store is loaded form $4978IS0 and the control store is loaded from $4978CS0 for all of the 4978 displays defined in the supervisor.

If a 4978 already has the stores loaded from a previous IPL sequence, the stores will not be reloaded. The combination of $4978IS0 and $4978CS0 provides an uppercase alphabet. The combination of $4978IS1 and $4978CS1 provides both upper and lower case alphabets.

If you have a keyboard other than a D02056 (RPQ) as the $SYSLOG device, the following procedure is necessary for installing EDX:

1. Using the stand-alone 4978 diskette, load the stores on the 4978 display (which corresponds to $SYSLOG).

2. IPL the starter supervisor. If the stores have already been loaded in the 4978 during a previous IPL sequence, the D02056 stores will not be reloaded.

3. Run the $TERMUT2 utility. Read the image and control stores into data sets $4978IS0 and $4978CS0 respectively. These data sets are on the IPL volume.

4. The starter system is now ready to be used with your keyboard.

Terminal Initialization for the Starter System

If your system includes a 4979 Display Station at device address 4, the starter system defines the program function keys as follows

KEY LABEL	FUNCTION
PF1	PF1
PF2	PF3
PF3	PF5
PF4	PF2
PF5	PF4
PF6	PF6

System Generation without the Program Preparation Facility

For Series/1 systems that do not include the Program Preparation Facility, installation requires the following general steps:

1. Assemble and link edit the supervisor, for the target Series/1 on a system that supports program preparation.
2. Assemble application programs for the target Series/1 on a system that supports program preparation.
3. Use utility program $INITDSK to initialize one or more diskettes with IPL text, space for the supervisor program, and a library to contain your application programs.
4. Transfer your supervisor to $EDXNUC on diskette(s) with either $COPY or $COPYUT1.
5. Copy $LOADER, any of the utilities, and the application programs that will be required on the target Series/1, onto the diskette(s) with $COPYUT1.
6. Install the diskette(s) on the target machine for execution.

```
          SYSTEM     STORAGE=256,                                    C
                MAXPROG=(3,1,5,2,2,1,1,4),                           C
                PARTS=(15,4,21,13,17,11,8,23)
          DISK       DEVICE=4963-64,ADDRESS=48,                      C
                VOLSER=EDX002,VOLORG=0,VOLSIZE=46,                   C
                LIBORG=129
          DISK       DEVICE=4963-64,VOLSER=EDX003,                   C
                BASEVOL=EDX002,VOLORG=46,                            C
                VOLSIZE=46,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=ASMLIB,                   C
                BASEVOL=EDX002,VOLORG=92,                            C
                VOLSIZE=45,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=EDX004,                   C
                BASEVOL=EDX002,VOLORG=138,                           C
                VOLSIZE=46,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=EDX005,                   C
                BASEVOL=EDX002,VOLORG=184,                           C
                VOLSIZE=46,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=EDX006,                   C
                BASEVOL=EDX002,VOLORG=230,                           C
                VOLSIZE=46,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=EDX007,                   C
                BASEVOL=EDX002,VOLORG=276,                           C
                VOLSIZE=46,LIBORG=1
          DISK       DEVICE=4963-64,VOLSER=EDX008,                   C
                BASEVOL=EDX002,VOLORG=322,                           C
                VOLSIZE=36,LIBORG=1
          DISK       DEVICE=4964,ADDRESS=02,VERIFY=NO
          DISK       DEVICE=4966,ADDRESS=22,VERIFY=NO,END=YES
$SYSLOG   TERMINAL   DEVICE=4979,ADDRESS=04,                        C
                HDCOPY=$SYSPRTR
$SYSLOGA  TERMINAL   DEVICE=TTY,ADDRESS=00,CRDELAY=4                 C
                PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR  TERMINAL   DEVICE=4974,ADDRESS=01,END=YES
          ENTRY      $EDXPTCH
$EDXPTCH  DATA       128F'0'              SYSTEM PATCH AREA
          STOREMAP
          END
```

Figure 17. Example of $EDXDEF: Configuration for 4963-64 (64MB disk) with a mapping of all (358) available cylinders

```
         SYSTEM   STORAGE=96,MAXPROG=(3,4),                C
               PARTS=(16,18),COMMON=START
         DISK     DEVICE=4963-58,ADDRESS=48,               C
               VOLSER=EDX002,VOLORG=0,VOLSIZE=46,          C
               LIBORG=129,FHVOL=FHVOL
         DISK     DEVICE=4963-58,VOLSER=EDX003,            C
               BASEVOL=EDX002,VOLORG=46,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4963-58,VOLSER=ASMLIB,            C
               BASEVOL=EDX002,VOLORG=92,                   C
               VOLSIZE=46,LIBORG=1
         DISK     DEVICE=4964,ADDRESS=02
         DISK     DEVICE=4966,ADDRESS=22,END=YES
$SYSLOG  TERMINAL DEVICE=4979,ADDRESS=04,                  C
               HDCOPY=$SYSPRTR
$SYSLOGA TERMINAL DEVICE=TTY,ADDRESS=00,CRDELAY=4,         C
               PAGSIZE=24,BOTM=23,SCREEN=YES
$SYSPRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES
         ENTRY    $EDXPTCH
$EDXPTCH DATA     128F'0'          SYSTEM PATCH AREA
         END
```

Figure 18. Example of $EDXDEF: Configuration for 4963-58 (58MB fixed-head disk)

CHAPTER 8. OVERVIEW OF THE INDEXED ACCESS METHOD

The Indexed Access Method licensed program is a data management facility that operates under the Event Driven Executive. It allows you to build, maintain, and access indexed data sets. 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 access to your records.

The Indexed Access Method offers the following 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 can be distributed throughout the data set and in a free pool at the end of the data set so that new records can be inserted. The space occupied by a deleted record is immediately available for new records.

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

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

- A utility program ($IAMUT1) which allows you to create, format, load, unload, and reorganize an indexed data set.

- A utility program ($VERIFY) which verifies the integrity of an indexed data set and reports on space utilization.

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

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

- The exclusive option - specifies that the file is for the exclusive use of a requester.
- Record locking - automatically prevents two requests from accessing the same data record at the same time.
- Immediate write back - causes all updated records to be written back to the file immediately.
- Accidental key modification is prevented - this helps ensure that your index matches the corresponding data.

DEVICES SUPPORTED

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

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

FUNCTIONS

Functions available include those that can be called from an application program and a utility to define and maintain an indexed data set.

I/O Requests

I/O requests allow you to build an indexed data set and to perform direct or sequential processing on that data set. Routines using these functions are written in Event Driven Language and can be included in programs written in any language that supports the calling of Event Driven Executive Language routines.

You request the services of the Indexed Access Method through the Event Driven Language CALL instruction in the following general form:

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

For information on coding the parameters and functions, refer to the Language Reference.

The following requests can be invoked:

Operands	Description
PROCESS	Builds an Indexed Access Control Block (IACB) and connects it to an indexed data set. You can then use the IACB to issue requests to that data set to read, update, insert, and delete records. A program can issue multiple PROCESS functions to obtain multiple IACBs for the same data set, enabling the data set to be accessed by several requests concurrently within the same program.
LOAD	Similar to PROCESS but used to load or extend 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. Use GET to retrieve a single record from the data set.
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. Use GETSEQ when you are performing sequential operations.
PUT	Loads or inserts a new record depending on whether the data set was opened with the LOAD or PROCESS request. Use PUT when you are adding records to a data set.
PUTUP	Replaces a record that is being held for update. Use PUTUP to modify a record.
PUTDE	Deletes a record that is being held for update. Use PUTDE to delete a record.
RELEASE	Releases a record that is being held for update. Use RELEASE when a record that was retrieved for update is not changed.
DELETE	Deletes a single record, identified by its key, from the data set. Use DELETE to delete a record; unlike PUTDE, the record cannot have been retrieved for update.
ENDSEQ	Terminates sequential processing.

EXTRACT Provides information about the file (from the File Control Block).

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

The $IAMUT1 Utility

The $IAMUT1 utility can be used to allocate, format, load, unload, or reorganize an indexed data set. Indexed Access Method requests can be used only on data sets defined either by this utility or by the Realtime Programming System Indexed Access Method. ($IAMUT1 is described in the Utilities, Operator Commands, Program Preparation, Messages and Codes manual.)

The $VERIFY Utility

The $VERIFY utility verifies the integrity of an indexed data set, and produces a report showing how the data set is defined and how the space is utilized. $VERIFY is described in the Utilities, Operator Commands, Program Preparation, Messages and Codes manual.

OPERATION OF THE INDEXED ACCESS METHOD

The Indexed Access Method performs I/O operations by using standard data management requests.

A single copy of the Indexed Access Method load module $IAM serves the entire system. It can be loaded automatically at IPL time through the automatic initialization capability (refer to "Automatic Application Initialization and Restart" on page 129), or it can be loaded manually by using the $L operator command. However, since the link module loads $IAM automatically, $IAM does not need to be loaded before it is used by any program. Once loaded, the Indexed Access Method remains in storage until cancelled with the $C operator command.

$IAM can be loaded into any partition, including partition one. It can be invoked (through the link module) from any partition, including the partition it is in. Figure 20 on page 149 shows an example of a system containing the Indexed Access Method.

INDEXED DATA SETS - OVERVIEW

You can organize a collection of data into an indexed data set if the data consists of fixed-length records and if each record can be uniquely identified by the contents of a single predefined field called the key. In an indexed data set, the records are arranged in ascending order by key. Reserved space, called free space, can be distributed throughout the data set so that records can be inserted.

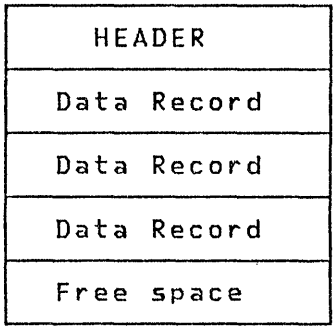
The total amount of free space for inserts is specified to the $IAMUT1 utility when the indexed data set is built. This free space is distributed throughout the data set in the form of free records within each data block, free blocks within each block grouping, and/or in a free pool at the end of the data set.

If you do not have any base records to load into your indexed data set in LOAD mode, you can define a "dynamic" data set which does not reserve space for records to be loaded. Such a data set has a dynamic structure which adjusts itself as required when records are inserted in PROCESS mode.

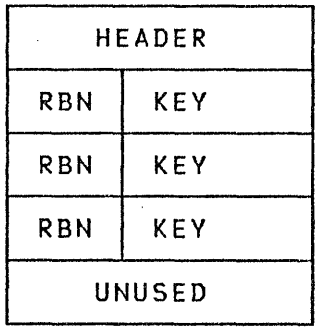
Data Set Format

Indexed data sets consist of data blocks which contain records, indexes (pointers) to the data blocks, and indexes to the index blocks. This technique is called a cascading index structure. The first two blocks in the indexed data set are the file control block (FCB), and its extension, which describe the attributes of the data set.

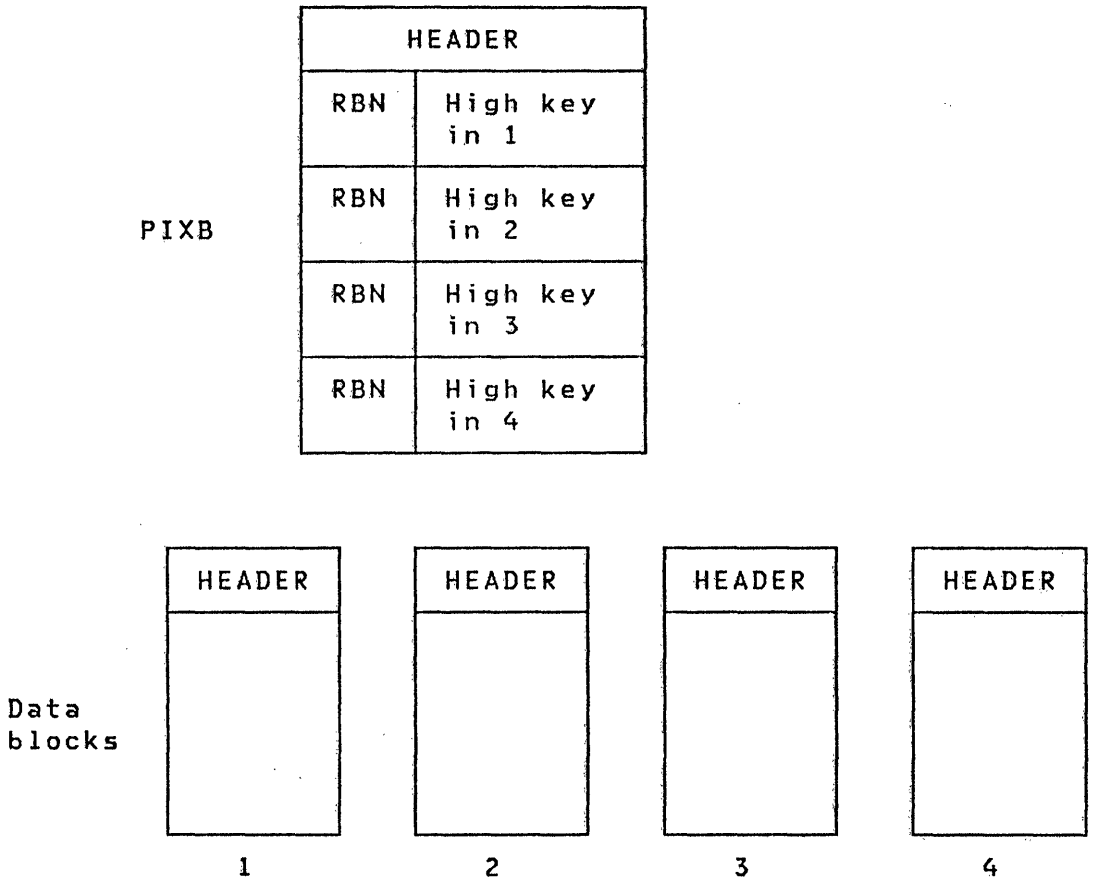
Each data block has the following format:



Each index block has the following format:



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



A Sample Cluster

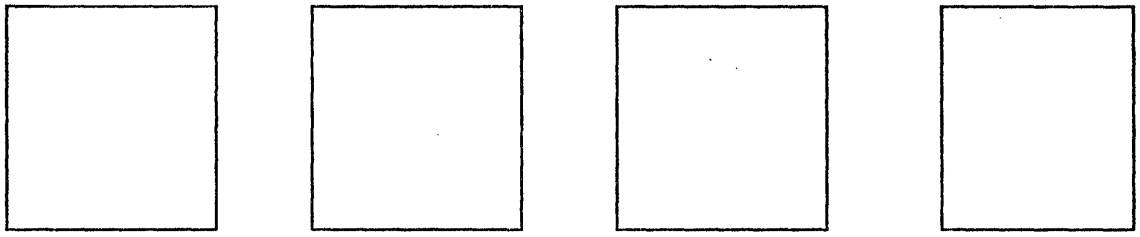
The records in each data block are in ascending order, according to the key field in each record.

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

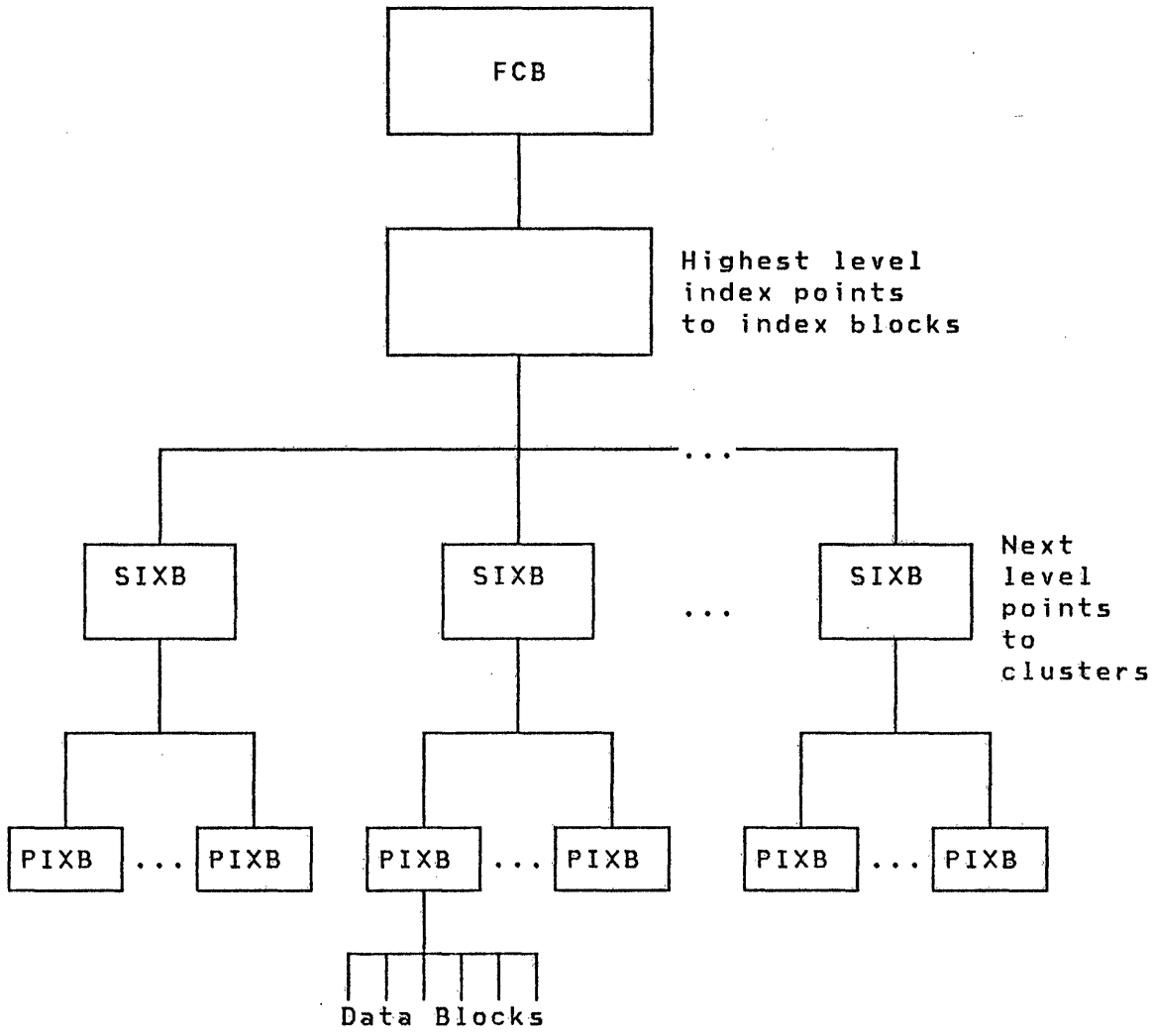
Each PIXB (or cluster) has an entry in a second-level index block (SIXB) that contains the address of the PIXB and the highest key in the cluster. The SIXB has the following structure:

SIXB

HEADER	
RBN	High key in PIXB1
RBN	High key in PIXB2
RBN	High key in PIXB3
RBN	High key in PIXB4



The SIXBs in the data set are described by an index block in the same manner as the PIXB describes each cluster. There is, of course, an index block that describes the entire dataset. The logical structure of the file is as follows:



Note that only the highest key in any data block is found in a PIXB entry, a SIXB entry contains only the highest key found in a PIXB, and so on, to the highest index block. This index technique is called **sparse indexing**.

REQUESTING RECORDS

When you request a record from your data set, the access method uses the index to retrieve the data block that contains the record. The index blocks and data blocks are read, using EDL READ instructions, into the central buffer. When the requested record is found, it is moved to the address you specified and control is returned to your program.

To minimize accesses to the disk, the buffer management algorithm tends to keep in the buffer the most frequently referenced blocks (index or data).

PREPARING TO EXECUTE INDEXED APPLICATIONS

The Indexed Access Method consists of the following components:

- A load module, $IAM, that supports the execution of the programs that contain your Indexed Access Method requests.
- A set of object modules that you may use to generate a customized load module. If you use the supplied load module, $IAM, you do not need the object modules.

  The object module, IAM, is called a **link module**. You include IAM with your program to provide the interface to the Indexed Access Method. This link module is sometimes called a **stub**.
- Two copy code modules, IAMEQU and FCBEQU. IAMEQU provides symbolic parameter values for constructing CALL parameter lists. FCBEQU provides a map of the file control block (FCB), and the FCB extension block.
- A load module for each of the Indexed Access Method utilities $IAMUT1 and $VERIFY.

Preparing Programs

To prepare an application programs that issues Indexed Access Method requests, perform the following steps:

1. Enter the source program, using one of the text editors ($FSEDIT, $EDIT1, or $EDIT1N).
2. Create the $LINK control statements required to combine your program with IAM (the link module) and any other object modules you may need in your application. These statements consist of a single OUTPUT statement, at least two INCLUDE statements - one for your program and one for IAM (the link module), and a single END statement. Use one of the text editors to perform this operation.
3. Assemble the source program using:

   The EDL compiler, $EDXASM, of the Program Preparation

Facility

or

The Series/1 macro assembler, $S1ASM, in conjunction with the Macro Library

or

The Series/1 macro assembler supplied by the System/370 Program Preparation Facility in conjunction with the Macro Library/Host

4. Use the linkage editor, $LINK, to combine the object modules into a single module, using the control statements prepared in Step 2.

5. Use the object program converter, $UPDATE or $UPDATEH, to convert your module to a loadable program.

When the preceding steps are completed, the program is ready to be executed.

Establishing the Data Set

Use the following steps to prepare the input for an indexed data set:

1. If your data records are 72 bytes or less use one of the text editors to enter your data or one of the communications utilities to get the data to your system. In either case, you must know the record format used by the utility. The utilities put two 80-byte records in each 256-byte EDX record. The first record begins at location 1, and the second record begins at location 129. The $IAMUT1 utility assumes unblocked input. $IAMUT1 takes only one logical record, the size of which was specified on the RECSIZE prompt, from each EDX record. Any record after the first logical record in each 256-byte EDX record is ignored. If you use the text editors, you must enter data on every other line starting with the first line.

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

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

Only one LOAD request can be issued to a data set at any time. Other processing requests can be made to a data set that is being loaded, but an attempt to retrieve a record from the data block being loaded can result in a no-record-found condition.

It is possible to define a "dynamic" indexed data set into which data records can never be loaded sequentially. You add records to such a file by inserting them in PROCESS mode. However, if you have initial base records, you should not dynamic file, since loading them sequentially in LOAD mode will result in a more efficient data set structure.

PROCESSING

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

- Direct reading - Retrieving a single record independently of any previous request.
- Sequential reading - Retrieving the next logical record relative to the previous request.
- Direct updating - Retrieving a single record for update; complete the update by either replacing or deleting the record.
- Sequential updating - Retrieving the next logical record for update; complete the update by either replacing or deleting the record.
- Inserting - Placing a single record, in its logical key sequence, into the indexed data set.
- Deleting - Removing a single record from the indexed data set.
- Extracting - Extracting data that describes the data set.

Note that the update functions require more than one request.

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

Direct Reading

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

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

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

Direct Updating

To update a record using direct access:

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

2. Modify the record in your buffer. Do not change the key field in the record. Return the updated record to the data set with a PUTUP request.

You can delete the record with a PUTDE request or leave it unchanged by issuing a RELEASE request.

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

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

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

Sequential Reading

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

Deleting

Use DELETE to delete a record from the data set. The full key of the record must be specified. If no record exists with the specified key, an error is indicated.

Deletion can also be performed as part of updating by following a GET for update with a PUTDE request.

Extracting

The EXTRACT request provides 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 data regarding the data set structure.

Execution of the EXTRACT request causes the file control block to be copied to an area that you provide. The EXTRACT request can also be used to copy the file control block extension to the area you provide. The extension contains a copy of the parameters that were used to define the indexed data set. The data set must have been connected by a LOAD or PROCESS request.

The contents of the FCB and its extension are described by FCBEQU, a unit of copy code that is supplied by the access method. Use COPY FCBEQU to include these equates in your program.

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 it to the printer. These procedures are provided by a combination of Event Driven Executive and Indexed Access Method services as suggested below. The Indexed Access Method utility $IAMUT1 does provide 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 making 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, copy the backup version of REPORT to the desired volume, and process the journal file to recreate the data set.

If a data-set-shut-down condition exists, IPL again. Then issue a PROCESS to the REPORT data set and, using the journal file, reprocess the transactions that occurred after 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 that were in process at the time of the problem is uncertain.

To recreate your data set, follow the steps in "Reorganization" to reorganize your data set. After recreating the data set, verify the status of the requests that were in process at the time the problem occurred.

Reorganization

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

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 $IAMUT1 utility to define a new indexed data set. Estimate the number of base records and the amount and mix of free space in order to minimize the need for future reorganizations. Refer to "The Indexed Data Set" on page 182 for guidelines for making these estimates.

3. Use the reorganize command (RO) of the $IAMUT1 utility 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 $IAMUT1 utility 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.

4. Use system utilities to delete the old data set and rename the new data set.

Dumping

To print records, use the DP command of the $DISKUT2 utility. $DISKUT2 produces 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, Program Preparation, Messages and Codes.

Deleting

Delete an indexed data set the same way you delete any other data set. From a terminal, use the DE command of the $DISKUT1 utility (refer to Utilities, Operator Commands, Program Preparation, Messages and Codes), or from a program use the $DISKUT3 data management utility (refer to "Chapter 16. Advanced Topics" on page 309).

Verifying

Use the $VERIFY utility to verify the accuracy of an indexed data set, and to provide information about its structure and use of free space. With $VERIFY you can:

- Print a formatted File Control Block (FCB) lisitng, including the FCB Extension block. This Extension block contains the original definition parameters for the indexed data set. Note that the FCB Extension block does not exist and definition parameters were not saved in the FCB prior to Version 1, Modification Level 2 of the Indexed Access Method
- Validate all pointers in any indexed data set
- Verify that the relationship between keys in the entries in the index blocks, and the high keys in the data blocks is correct
- Print the amount of free space in your data set, which may indicate a reorganization is needed

CONCATENATING DATA SETS

The ALTIAM subroutine allows you to concatenate multiple IAM data sets and to issue normal IAM commands to the concatenated file. This allows you to have more than 32,767 sectors in an IAM file or to put parts of a file on different devices to improve performance. The data sets may reside on the same or different volumes or devices. The keys of all data sets must have the same location and length. Each file must be loaded individually and have a unique range of keys, with no overlap of key ranges between the data sets.

To incorporate this function in your application, transcribe the ALTIAM subroutine using one of the text editors and modify it to meet your requirements. Compile it with $EDXASM or the Series/1 Macro Assembler and add the object program to your

object library. Include the object program when you link edit your application programs with the IAM link module.

Note: The ALTIAM subroutine is not compatible with the Multiple Terminal Manager.

The ALTIAM subroutine accepts all Indexed Access Method requests for single files. A special request, CONCAT, is issued to concatenate files. Only one set of files may be concatenated per copy of ALTIAM; when the file is disconnected, another set may be concatenated. The parameters to CONCAT are as follows:

```
CALL  ALTIAM,(CONCAT),IACB,(DSCBTAB),(OPENTAB),(MODE)
```

- Equate CONCAT to 14.
- IACB, OPENTAB, and MODE are the same as in the PROCESS request.
- DSCBTAB is the address of a list of opened data set control blocks (DSCBs) with the following format:

```
DSCBTAB     DATA  A(DS1)
            DATA  A(DS2)
            DATA  A(DS3)
            DATA  A(BUFFER)
```

The DSCBs must be in order of increasing key ranges of of the corresponding files. Three DSCBs is the default but you may increase or decrease the number. If only two data sets are needed, word three must be zero. The buffer must be large enough to hold the largest record in the concatenated file.

The CONCAT function issues PROCESS requests and reads the low key of each file. The default maximum key size (50 bytes) may be changed. The address of the IACB that is returned is used by ALTIAM to issue processing requests against the concatenated file.

The following requests may be made to a concatenated file:

```
GET
GETSEQ
PUT
PUTUP
PUTDE
DELETE
EXTRACT
ENDSEQ
RELEASE
DISCONN
```

```
01390 *
01400 DEL      EQU   *
01410 **  PROCESS DELETE REQUESTS
01420          MOVE    #1,PARM3              POINT AT USERS KEY
01430          MOVE    COMPLEN,AKSIZE        FULL KEY SUPPLIED
01440          GOTO    CHECK
01450 *
01460 SEQ      EQU   *
01470 **  PROCESS GET SEQ REQUESTS
01480          IF      (ASEQ,EQ,1),GOTO,LAST IF NOT FIRST IN SEQUENCE
01490          MOVE    ASEQ,1                SIGNAL SEQUENTIAL MODE
01500 **  PROCESS FIRST SEQUENTIAL AS DIRECT
01510 *
01520 DIR      EQU   *
01530 **  PROCESS GET REQUESTS
01540          IF     (PARM4,EQ,0)           IF KEY IS NOT SET
01550            MOVEA  IIACB,ALTIACB        POINT AT FIRST FILE
01560            GOTO   INRANGE              SKIP CHECKING
01570          ENDIF
01580          MOVE   #1,PARM4              GET KEY POINTER
01590          MOVE   COMPLEN,(-1,#1),BYTE  GET KEY LENGTH
01600          SHIFTR COMPLEN,8             GET INTO POSITION
01610 *
01620 CHECK    EQU  *
01630 ********************************************************************
01640 ** LOOP THRU IACB TABLE COMPRING USERS KEY (#1) TO SAVED KEY IN
01650 ** THE TABLE.  THE SAVED KEY IS THE LOWEST KEY IN THE NEXT FILE.
01660 ********************************************************************
01670          MOVEA  #2,ALTIACB            POINT AT IACB TABLE
01680          MOVE   REGA,#1               SAVE USERS KEY ADDRESS
01690          DO     +DSCB#,TIMES          LOOP THRU IACBS
01700            IF     ((0,#2),EQ,0),GOTO,INRANGE      EXIT IF NO MORE
01710            MOVE   IIACB,#2              SAVE CURRENT IACB
01720            ADD    #2,2                  POINT AT SAVED KEY
01730            MOVE   COUNT,0               INITIALIZE STRING COUNTER
01740 *
01750            DO     WHILE,(COUNT,LE,COMPLEN)   LOOP THRU STRING
01760              IF    ((0,#1),LT,(0,#2),BYTE),GOTO,INRANGE CORRECT IACB
01770              IF    ((0,#1),GT,(0,#2),BYTE),GOTO,OUTRANGE WRONG IACB
01780              ADD    #1,1                 INCREMENT POINTERS
01790              ADD    #2,1                 * IF STRINGS ARE EQUAL
01800              ADD    COUNT,1
01810            ENDDO
01820 *
01830 **  IF STRINGS ARE EQUAL THEN THE KEY IS IN THE NEXT FILE.  UNLESS
01840 **  WE ARE USING THE LAST FILE ALREADY.
01850            ADD    IIACB,+AENTSIZE,RESULT=#2  POINT AT NEXT
01851            MOVE   DOUBLE1,0
01852            MOVE   DOUBLE2,#2
01860            IF     (DOUBLE1,LT, ALSTIACB,DWORD) IF NOT THE LAST IACB
01870              MOVE   IIACB,#2           STORE NEW POINTER
01880            ENDIF
01890            GOTO   INRANGE             FOUND THE CORRECT IACB
```

```
01900 *
01910 OUTRANGE  EQU   *
01920 **  KEY IS NOT IN THIS RANGE. CHECK THE NEXT.
01930             ADD    IIACB,+AENTSIZE,RESULT=#2  BUMP THE IACB POINTER
01940             MOVE   #1,REGA             RESTORE THE USER KEY POINTER
01950           ENDDO
01960 *
01970 INRANGE   EQU   *
01980 **  KEY IS IN THIS RANGE.  ISSUE THE IAM CALL.
01990           CALL   CALLIAM
02000 *
02010           IF     (REGA,EQ,-58),AND,(PARM5,GT,+UPEQ)  NO RECORD FOUND
02020             ADD    IIACB,+AENTSIZE    POINT AT NEXT IACB
02021             MOVE   DOUBLE1,0
02030             MOVE   DOUBLE2,IIACB       IN A REGISTER
02031             MOVE   #1,DOUBLE2
02040             IF     (DOUBLE1,LT, ALSTIACB,DWORD),AND, IN RANGE        X
02050                ((0,#1),NE,0),GOTO,INRANGE    * TRY NEXT FILE
02060           ENDIF
02070           GOTO   EXIT
02080           EJECT
02090 ************************************************************************
02100 **  INVOKE IAM AND SAVE RETURN CODE.
02110 ************************************************************************
02120           SUBROUT CALLIAM
02130           MOVE   ALSTIACB+2,IIACB      UPDATE LAST IACB CELL
02140           CALL   IAM,+PROCESS,IACB,(IACB),(IACB),+EQ,P2=IFUNC,        X
02150                P3=IIACB,P4=IBUFF,P5=IKEY,P6=IOPT
02160           MOVEA  TCW,$TCBCO-$TCB#1    OFFSET TO TASK CONTROL WORD
02170           MOVE   REGA,#1,P2=TCW       PICK UP TASK CONTROL WORD
02180           RETURN
02190           SPACE 5
02200 ALTEOD    EQU   *
02210 ************************************************************************
02220 **  END OF DATA EXIT.  IF NOT THE LAST FILE SWITCH TO THE NEXT ONE.
02230 **  IF THE LAST FILE PASS CONTROL TO USERS EOD EXIT.
02240 ************************************************************************
02250           ADD    IIACB,+AENTSIZE         POINT TO THE NEXT IACB
02251           MOVE   DOUBLE1,0
02252           MOVE   DOUBLE2,IIACB       IN A REGISTER
02260           MOVE   #1,IIACB
02270           IF     (DOUBLE1,LT, ALSTIACB,DWORD),AND, IN RANGE          X
02280                ((0,#1),NE,0)
02290             MOVE   IKEY,0               GET FIRST KEY IN NEXT FILE
02300             GOTO   INRANGE              ISSUE IAM REQUEST
02310           ENDIF
02320 *
02330         MOVE    ASEQ,0                  RESET SEQUENTIAL SWITCH
02340         IF      (AEOD,NE,0)             IF END OF DATA EXIT EXISTS
02350           GOTO    (AEOD)                GO TO IT
02360         ELSE
02370           GOTO    EXIT
02380         ENDIF
02390           SPACE 5
```

If insert activity is to be primarily into one or more areas or key ranges, however, 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 $IAMUT1 utility.

Using Dynamic Data Sets

Estimating free space requires a considerable amount of knowledge about the data that will be placed in your data set, and careful planning to define the characteristics of the data set. The estimating free space procedure results in a well structured file with efficient operation.

Defining a Dynamic File

In some cases, you may not be able to predict the type of processing that will be done on a data set. The Indexed Access Method has the capability to adjust a data set dynamically according to the needs of the processing. For this reason you need not specify the FREEREC, FREEBLK, RSVBLK, RSVIX, or FPOOL parameters using the SE command of the $IAMUT1 utility. Instead, you can specify the actual number of blocks to be assigned to the free pool by the DYN parameter. This is especially useful when you have no (or relatively few) base records to load, but will place all or most records into the data set by inserting them in random sequence. (When such a data set has grown to its working size, it should be reorganized for more efficient operation.)

Defining a Free Pool

You can specify the DYN parameter in conjunction with other free space parameters. Before you load base records or reorganize a data set, it is likely that you will want to specify the FREEREC and FREEBLK parameters to provide structured free space throughout the data set. You can also specify the DYN parameter to provide free pool blocks. This allows dynamic restructuring of those portions of the data set affected if any part of the structured free space becomes full.

Defining an Expanded Free Pool

You can also specify the DYN parameter in conjunction with the RSVBLK, RSVIX, and FPOOL parameters. In this case, the amount of free pool space specified by the DYN parameter is in addition to that provided by the FPOOL parameter.

Dynamic Data Set Processing

In order to provide a full dynamic capability, the Indexed Access Method can restructure a data set in two ways:

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

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

Converting to a Dynamic Data Set

A data block can become empty only if the delete threshold (DELTHR) parameter is zero. Previous versions of the Indexed Access Method would not allow a value of zero, and would internally reset it to a non-zero value if zero was specified. This version (1.2) of the Indexed Access Method allows a value of zero and retains it internally if specified. The reorganize (RO) $IAMUT1 command can be used to activate all new Indexed Access Method functions for indexed data sets built with previous versions of the Indexed Access Method. In this version of the Indexed Access Method, the DELTHR parameter defaults to zero if the DYN parameter is specified.

Specifying the DYN parameter

When you specify the number of blocks for the DYN parameter, remember that the Indexed Access Method can store several of your data records in a block, depending on the record size and block size you specify. Each block contains a 16 byte header, thus the number of records that can be contained in each block can be calculated by the following formula

$$\text{Records per block} = \frac{(\text{BLKSIZE}-16)}{\text{RECSIZE}}$$

(Use integer quotient only; discard remainder)

Note that blocks can be taken from the free pool for use as index blocks as well as for data blocks, so provide some extra blocks for these. A reasonable estimate of the number of index blocks required is 10%. Thus, if you know the number of data records you would like to add to the file, you can calculate the number of blocks to specify for the DYN parameter as follows

$$DYN = \frac{(\text{Number of records to insert}) \times 1.1}{(\text{Records per block})}$$

Building The Indexed Data Set

The SE and DF commands of the $IAMUT1 utility allow you to specify the size and format of your indexed data set and to format the data set. Use the SE command to enter those values that determine the size of the indexed data set and to receive a display of the size calculation information. Use the DF command to format the data set, using the values previously specified on the SE command.

Determining Size and Format

The structure of the data set is determined by the following parameters of the SE command:

- BASEREC - Estimated number of base records
- BLKSIZE - Block size
- RECSIZE - Record size
- KEYSIZE - Key size
- KEYPOS - Key position
- 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
- DYN - Number of blocks to add to free pool

The define (DF) command fixes the size of the data set. Therefore, BASEREC, FREEREC, FREEBLK, RSVBLK, RSVIX, FPOOL, and DYN 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 defined parameters, use the SE command.

The DF command allows you to select the immediate write-back option. If you select this option, modified records are written to the file immediately; this contributes to the integrity of the file; however, response time increases.

Defining and Creating the Indexed Data Set

The setparms (SE) command allows you to review the size calculation information without actually formatting the data set. $IAMUT1 returns to your terminal the size of the data set and other information. The calculations performed by the SE function are described in "Data Set Format" on page 192.

Use the DF command to format the data set. You are prompted for the volume and data set names and the immediate write-back option. (Note: the data set must have been previously created using the CR command of the $IAMUT1 utility or the AL command of the $DISKUT1 utility.) The 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, $IAMUT1 returns the amount of space required. 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 occurs and $IAMUT1 gives the option to retry. If you retry, the utility prompts for the volume and data set names, and the function is attempted again.

Using the $IAMUT1 Utility - Examples

A data set is to accommodate 10,000 base records with a record size of 70 bytes. An estimated 5,000 records are to be inserted.

Selecting a block size of 256 allows three records per block ((256-16)/70)) with a remainder of 30 bytes. If the data set were created with one free record per block, the ratio of two

base records to one free record would accurately reflect the insert activity. Buffer size is minimized. Some space (30 bytes per block) is wasted.

Selecting a block size of 512 allows seven records per block ((512-16)/70) with a remainder of six bytes. If the data set were created with two free records per block, the ratio of five base records to two free records would overestimate the insert activity. The larger block size requires a larger buffer but increases I/O efficiency. In addition, fewer bytes are wasted (six bytes).

Assume that the user has entered the DF subcommand to allocate the file using the specifications shown in Example 2. Name the file IDATA and placed it on EDX002.

Example 1

```
ENTER COMMAND (?): SE
PARAMETER    DEFAULT  NEW VALUE
BASEREC          NULL :10000
BLKSIZE             0 :256
RECSIZE             0 :70
KEYSIZE             0 :10
KEYPOS              1 :1
FREEREC             0 :1
FREEBLK             0 :0
RSVBLK           NULL :0
RSVIX               0 :0
FPOOL            NULL :0
DELTHR           NULL :0
DYN              NULL :0
TOTAL LOGICAL RECORDS/DATA BLOCK:                3
FULL RECORDS/DATA BLOCK:                         2
INITIAL ALLOCATED DATA BLOCKS:                5000
INDEX ENTRY SIZE:                               14
TOTAL ENTRIES/INDEX BLOCK:                      17
FREE ENTRIES/PIXB:                               0
RESERVE ENTRIES/PIXB(BLOCKS):                    0
FULL ENTRIES/PIXB:                              17
RESERVE ENTRIES/SIXB:                            0
FULL ENTRIES/SIXB:                              17
DELETE THRESHOLD ENTRIES:                        0
FREE POOL SIZE IN BLOCKS:                        0
# OF INDEX BLOCKS AT LEVEL   1:                295
# OF INDEX BLOCKS AT LEVEL   2:                 18
# OF INDEX BLOCKS AT LEVEL   3:                  2
# OF INDEX BLOCKS AT LEVEL   4:                  1
DATA SET SIZE IN EDX RECORDS:                 5318
INDEXED ACCESS METHOD RETURN CODE:              -1
SYSTEM RETURN CODE:                             -1
```

Example 2

```
ENTER COMMAND (?): SE
PARAMETER  DEFAULT NEW VALUE
BASEREC      10000 :
BLKSIZE        256 :512
RECSIZE         70 :
KEYSIZE         10 :
KEYPOS           1 :
FREEREC          1 :2
FREEBLK          0 :
RSVBLK           0 :
RSVIX            0 :
FPOOL            0 :
DELTHR           0 :
DYN              0 :
TOTAL LOGICAL RECORDS/DATA BLOCK:                    7
FULL RECORDS/DATA BLOCK:                             5
INITIAL ALLOCATED DATA BLOCKS:                    2000
INDEX ENTRY SIZE:                                   14
TOTAL ENTRIES/INDEX BLOCK:                          35
FREE ENTRIES/PIXB:                                   0
RESERVE ENTRIES/PIXB(BLOCKS):                        0
FULL ENTRIES/PIXB:                                  35
RESERVE ENTRIES/SIXB:                                0
FULL ENTRIES/SIXB:                                  35
DELETE THRESHOLD ENTRIES:                            0
FREE POOL SIZE IN BLOCKS:                            0
# OF INDEX BLOCKS AT LEVEL    1:                    58
# OF INDEX BLOCKS AT LEVEL    2:                     2
# OF INDEX BLOCKS AT LEVEL    3:                     1
DATA SET SIZE IN EDX RECORDS:                     4126
INDEXED ACCESS METHOD RETURN CODE:                  -1
SYSTEM RETURN CODE:                                 -1
```

Note: Respond to the prompts with the values you wish to change. The utility reuses the values from previous execution.

Example 3

```
ENTER COMMAND (?): DF
DO YOU WANT IMMEDIATE WRITE-BACK? N
ENTER DATA SET (NAME,VOLUME): IDATA,EDX002
TOTAL LOGICAL RECORDS/DATA BLOCK:                    7
FULL RECORDS/DATA BLOCK:                             5
INITIAL ALLOCATED DATA BLOCKS:                    2000
INDEX ENTRY SIZE:                                   14
TOTAL ENTRIES/INDEX BLOCK:                          35
FREE ENTRIES/PIXB:                                   0
RESERVE ENTRIES/PIXB (BLOCKS):                       0
FULL ENTRIES/PIXB:                                  35
RESERVE ENTRIES/SIXB:                                0
FULL ENTRIES/SIXB:                                  35
DELETE THRESHOLD ENTRIES:                            0
FREE POOL SIZE IN BLOCKS:                            0
# OF INDEX BLOCKS AT LEVEL    1:                    58
# OF INDEX BLOCKS AT LEVEL    2:                     2
# OF INDEX BLOCKS AT LEVEL    3:                     1
DATA SET SIZE IN EDX RECORDS:                     4126
INDEXED ACCESS METHOD RETURN CODE:                  -1
SYSTEM RETURN CODE:                                 -1

ENTER COMMAND (?): EN

$IAMUT1  ENDED AT 00:38:47
```

The key differences between Example 1 and Example 2 are:

- Fewer records (256-byte blocks) are required for Example 2.
- The index in Example 2 is a three-level index, while in Example 1 it is a four-level index. This eliminates one disk access, improving performance slightly.
- Each data block has two free records in Example 2. In example 1 each data block has one free record.

Example 4 - Dynamic Data Set

```
ENTER COMMAND (?): SE
PARAMETER DEFAULT NEW VALUE
BASEREC       NULL :
BLKSIZE          0 :256
RECSIZE          0 :70
KEYSIZE          0 :10
KEUPOS           1 :
FREEREC          0 :
FREEBLK          0 :
RSVBLK        NULL :
RSVIX            0 :
FPOOL         NULL :
DELTHR        NULL :
DYN           NULL : 5300
TOTAL LOGICAL RECORDS/DATA BLOCK:              3
FULL RECORDS/DATA BLOCK:                       3
INITIAL ALLOCATED DATA BLOCKS:                 1
INDEX ENTRY SIZE:                             14
TOTAL ENTRIES/INDEX BLOCK:                    17
FREE ENTRIES/PIXB:                             0
RESERVE ENTRIES/PIXB(BLOCKS):                  0
FULL ENTRIES/PIXB:                            17
RESERVE ENTRIES/SIXB:                          0
FULL ENTRIES/SIXB:                            17
DELETE THRESHOLD ENTRIES:                      0
FREE POOL SIZE IN BLOCKS:                   5300
# OF INDEX BLOCKS AT LEVEL   1:               1
DATA SET SIZE IN EDX RECORDS:               5304
INDEXED ACCESS METHOD RETURN CODE:            -1
SYSTEM RETURN CODE:                           -1
```

Example 5 - Dynamic Data Set

```
ENTER COMMAND (?): DF
DO YOU WANT IMMEDIATE WRITE-BACK? N
ENTER DATA SET (NAME,VOLUME): IDATA,EDX002
TOTAL LOGICAL RECORDS/DATA BLOCK: 3
FULL RECORDS/DATA BLOCK: 3
INITIAL ALLOCATED DATA BLOCKS:                1
INDEX ENTRY SIZE:                            14
TOTAL ENTRIES/INDEX BLOCK:                   17
FREE ENTRIES/PIXB:                            0
RESERVE ENTRIES/PIXB(BLOCKS):                 0
FULL ENTRIES/PIXB:                           17
RESERVE ENTRIES/SIXB:                         0
FULL ENTRIES/SIXB:                           17
DELETE THRESHOLD ENTRIES:                     0
FREE POOL SIZE IN BLOCKS:                  5300
# OF INDEX BLOCKS AT LEVEL   1:               1
DATA SET SIZE IN EDX RECORDS:              5304
INDEXED ACCESS METHOD RETURN CODE:           -1
SYSTEM RETURN CODE:                          -1
```

Examples 4 and 5 show the SE and DF commands used to create a dynamic indexed data set. Note that the resulting data set has only one allocated data block and only one index block. The majority of the space is in the free pool as specified by the DYN parameter.

Data Set Format

The define command of the $IAMUT1 utility formats and creates an indexed data set.

Use the DF command to format the data set. You are prompted for the volume and data set names and the immediate write-back option. (Note: the data set must have been previously created using the CR command of the $IAMUT1 utility or the AL command of the $DISKUT1 utility.) The 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, $IAMUT1 returns the amount of space required. Knowing the available space and using the SE The information required to establish the format and the number of blocks in a data set is provided by ten parameters of the SE command.

Parameter	Definition
BASEREC	Number of base records
BLKSIZE	Block size
RECSIZE	Record size
KEYSIZE	Key size
KEYPOS	Key position
FREEREC	Number of free records per block
FREEBLK	Percentage of free blocks
RSVBLK	Percentage of reserved blocks
RSVIX	Percentage of reserved index
FPOOL	Percentage of free pool
DELTHR	Percentage of blocks to retain when deleting records
DYN	Number of blocks to add to free pool

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. Block size must be a multiple of 256. A block is addressed by its relative block number (RBN). The first block in the data set is located at RBN 0.

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

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

2. The RBN of the first block in an indexed data set is 0. The record number of the first Event Driven Executive 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 $IAMUT1 utility.

Four kinds of blocks exist in an indexed data set: a file control block (FCB), a file contol block extension, index blocks, and data blocks. These blocks are all the same length, as defined by BLKSIZE, but they contain different kinds of information. The FCB contains control information, the FCB extension contains saved file definition input parameters, index blocks contain index entries and data blocks contain data records. The control information is also contained in block headers; a description of control information is contained in Internal Design Figure 23 also shows examples of the four block types.

File control block extension

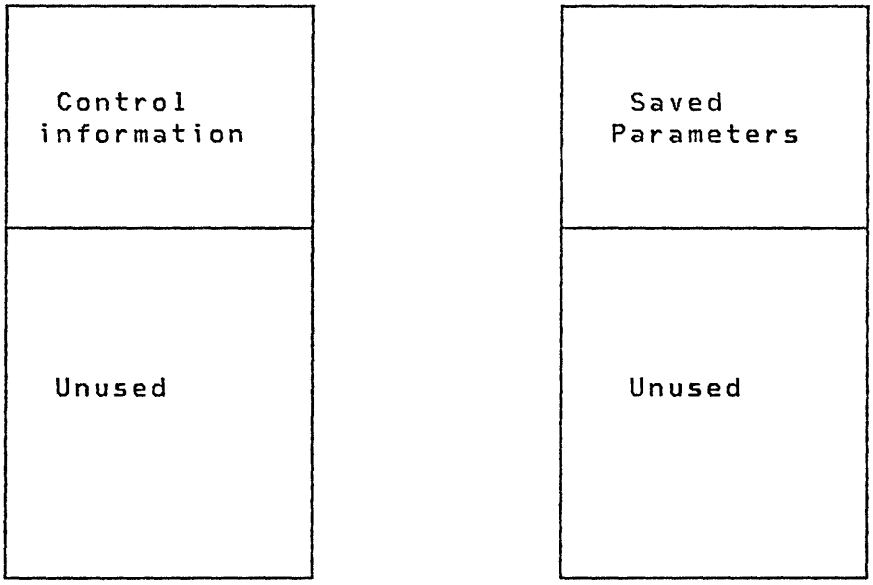


Figure 23 (Part 1 of 2). Indexed Data Set Block Types

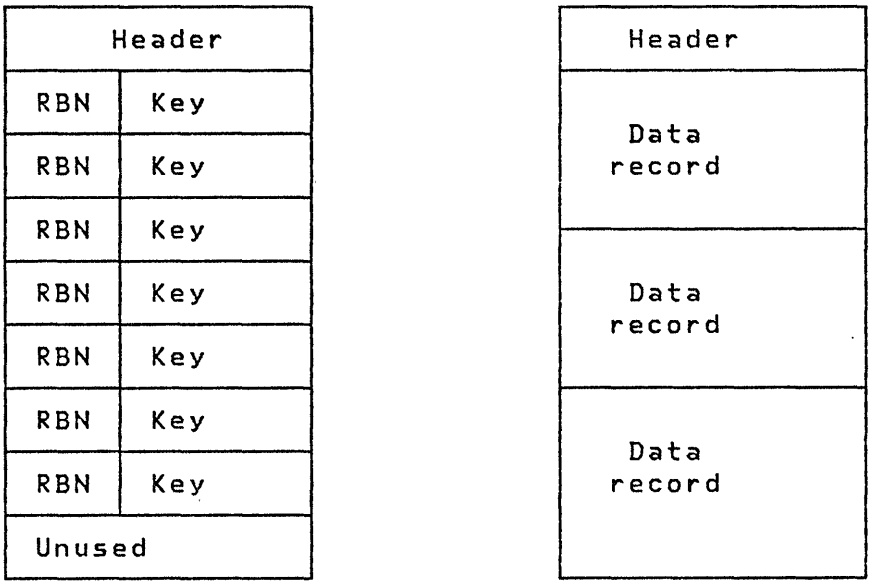


Figure 23 (Part 2 of 2). Indexed Data Set Block Types

The File Control Block

The file control block (FCB) is the first block in the data set (RBN 0); it contains control information. The field names in the FCB can be seen by examining a listing of FCBEQU, a copy code module that is supplied as part of the Indexed Access Method.

The File Control Block Extension

The file control block extension is the second block in the data set (RBN 1); it contains the saved file definition parameters as specified by the user. The field names in the FCB extension can also be seen by examining a listing of FCBEQU. The saved parameters can be refered to in either of two ways:

- From a program, via the EXTRACT function, or
- By running the $VERIFY utility, which prints the values.

Index Block

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

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

The result is truncated; any remainder represents the number of unused bytes in the block. For example, if block size is 256 and key length is 28, then each index entry is 32 bytes, there are 7 entries in a block, and the last 16 bytes of the block are unused.

Data Block

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

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

The result is truncated; any remainder represents the number of unused bytes in the block. For example, if block size is 256 and record size is 80, the data block can accommodate three records and there is no unused area. The key field of the last record slot in an index block is the high key for the data block. If some records of the data block are not currently used, the key field of the last record slot is the same as the key field of

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. Because rounding occurs in calculating the number of clusters, a few more allocated records than required may exist in the last allocated block. The last cluster can be a short one because only the required number of blocks are used.

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

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 the sequential chain. Chaining allows sequential processing of the data set with no need to reference the index. When a free block is converted to an allocated block, the free block is included in the chain.

Free Pool

If you specify that you want a free pool (with the FPOOL and/or DYN parameter of the SE command of the $IAMUT1 utility), your indexed data set contains a pool of free blocks. The file control block contain s a pointer to the first block of the free pool, and all blocks in the free pool are chained together by forward pointers.

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

When a data block becomes empty because of record deletions, the data block may return to the free pool (depending on the delete threshold (DELTHR) parameter). If the data block is

returned to the free pool, reference to the block is removed from the primary-level index block, and the block is placed at the beginning of the free pool chain. Index blocks are never returned to the free pool.

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

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

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

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

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

- 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, plus the number of blocks specified by the DYN parameter.

STORAGE AND PERFORMANCE

Storage Requirements

The minimum amount of storage required by the Indexed Access Method to perform all functions is about 15.2KB, not including the link module or any error exit routine you may have written. The storage estimate 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, a 512-byte buffer is required. If your maximum block size is larger than 256 bytes, the buffer size is twice your block

size. You can improve performance by making the buffer larger. The program directory that is shipped with your PID material contains a description of the size and capacity of the buffer and information on how to modify it. The buffer that is defined in $IAM should provide adequate performance for most applications.

- One user connected to an indexed file at a time. If more than one user is connected, add about 625 bytes per user.

The size of the IBM-supplied link module which is included in your application program 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. These parameters are described in "Data Set Format" on page 192.

Performance

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 (refer to "Data Set Format" on page 192). The following factors affect performance:

- 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 degrading 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.

- File organization

The dynamic file capability makes it easy for you to define and use files without planning the file structure. You should be aware that heavy use of the free pool (as occurs with a dynamic file) has an impact on performance.

The best performance from an indexed file is attained when the file structure is well planned and the free pool is rarely used, if it exists at all. This is because the high-level index blocks are concentrated in a single area. Thus, an access to the file requires only two significant seeks. This file structure is maintained as long as new records are inserted in the space provided by the FREEREC and FREEBLK parameters.

However, when the free pool is heavily utilized, the logical structure of the file is no longer reflected in the physical positions of the blocks. As a result, every block that must be read in order to to satisfy a request could result in a significant seek. This increase in the number of significant seeks results in an increase in the elapsed time required to process the request.

Use the $IAMUT1 utility to see the affects of the varicus parameters on the file structure. (Refer to "Using the $IAMUT1 Utility - An Example" on page 188 for an example.)

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 using the Indexed Access Method concurrently, resource contention can result, and performance is degraded.

The FTAB table provides the screen location (line and spaces) and size (characters) of each parameter field on the menu, in ascending order. The session manager program $SMCTL uses the FTAB table to retrieve the parameters it uses to replace the &PARMnn. fields before passing the procedure to $JOBUTIL. The parameter &PARM00. always represents your one to four character logon ID.

The &SAVEmm fields in the parameter part of the procedure point to fields in the parameter save data set $SMPnnnn (where nnnn is the logon ID) where the parameters you enter are saved from session to session. The two digits, mm, are used to index into the data set.

Note that multiple &PARMnn. fields between PARAMETER and END are sequential, beginning with $PARM01.

The following table lists the $SAVEmm fields, the procedure with which they are associated, and the utility or function invoked. When assigning values to the index digits (mm) in your procedure, start with 90 and work backwards to 61.

FIELD #	PROCEDURE	UTILITY/FUNCTION
$SAVE01-03	$SMP0201	$EDXASM
$SAVE04-06	$SMP0202	$S1ASM
$SAVE07-13	$SMP0203	$COBOL
$SAVE14-16	$SMP0204	$FORT
$SAVE17-18	$SMP0205	$LINK
$SAVE19-22	$SMP0206	$UPDATE
$SAVE23-24	$SMP0208	$PREFIND
$SAVE25-26	$SMP0308	$MOVEVOL
$SAVE27	$SMP0405	$FONT
$SAVE28	$SMP0501	$DIUTIL
$SAVE29	$SMP0502	$DICOMP
$SAVE30	$SMP0503	$DIINTR
$SAVE31-35	$SMP06	Execute application program
$SAVE36	$SMP0801	$BSCTRCE
$SAVE37	$SMP0806	$PRT2780
$SAVE38	$SMP0807	$PRT3780
$SAVE39	$SMP0808	$HCFUT1
$SAVE40-41	$SMP0208	$PREFIND
$SAVE42	$SMP0901	$TRAP
$SAVE43	$SMP0902	$DUMP
$SAVE44	$SMP0903	$LOG
$SAVE45-49	$SMP0210	$PLI
$SAVE50-60	Reserved	

```
PARAMETER
&PARM01,&SAVE01
&PARM02,&SAVE02
&PARM03,&SAVE03
END
LOG        OFF
REMARK    @ASSEMBLE &PARM01. TO &PARM02. USERID=&PARM00.
JOB        $SMP0201
PROGRAM    $EDXASM,ASMLIB
PARM       &PARM03.
DS         &PARM01.
DS         $SM1&PARM00.,EDX003
DS         &PARM02.
EXEC
EOJ
END
```

Figure 33. Invoking EDXASM

Parameters that have been saved are retrieved from the $SMPnnnn data set according to the relationships in the first part of the procedure. These parameters are displayed on the terminal. Then any parameters you enter from the terminal are used to update the procedure.

ALLOCATING AND DELETING WORK DATA SETS

The session manager allocates work data sets at logon time. They may be deleted at logoff time with one of the text editors. Two data sets, $SMALLOC and $SMDELET, are provided which are used in allocating and deleting data sets. $SMALLOC contains the data sets to be allocated and $SMDELET contains the data sets to be deleted. Figure 34 on page 223 lists the contents of $SMALLOC and Figure 35 on page 224 lists the contents of $SMDELET.

You may tailor the work data set allocations and deletions by modifying the $SMALLOC and $SMDELET data sets via the $FSEDIT utility. Modifications usually consists of changing the size or volume of a data set. However, you may allocate and delete up to four additional data sets. By moving the END terminator below $SM7 (statement 00120), you may allocate data sets $SM4, $SM5, $SM6, and $SM7. If you modify $SMALLOC, you should also modify $SMDELET to be consistent.

PROGSTOP instruction.

Using the Task Error Exit Facility in Your Task

Linkage Conventions

To make use of the Task Error Exit facility in your task, you must code a small control block and the error exit routine. In addition, you must set aside the block of storage that will be filled with the hardware status when an exception occurs.

The control block, called the task error exit control block (TEECB), provides the linkage between the supervisor and your error exit. The TEECB must be aligned on a fullword boundary.

To allow the supervisor to find your TEECB, you should code its address as the value of the ERRXIT keyword parameter of the PROGRAM or TASK EDL statement that defines your task.

The format of the TEECB is as follows:

```
TEECB    DS  0F          TASK ERROR EXIT CONTROL BLOCK
TEECTL   DC  X'0002'      ------------------------
TEESIA   DC  A(XITRTN)   |_____0_____|_____2_____|
TEEHSA   DC  A(HSA)      |___________SIA___________|
                         |___________HSA___________|
```

In the first word (TEECTL), bits 0 - 7 are reserved and must be zero. Bits 8-15 state the number of data words that follow. This value must be two. The second word (TEESIA) contains the address of the starting instruction of your Error Exit routine. The last word (TEEHSA) contains the address of the block of storage you have reserved to receive the hardware status when an exception occurs. This block is called the Hardware Status Area (HSA) and is 24 bytes long.

The format of the HSA is:

```
*       HARDWARE STATUS AREA
HSA       DS      0F     ALIGN ON FULL WORD BOUNDARY
HSAPSW    DS      1F     PROGRAM STATUS WORD
HSALSB    EQU     *      LEVEL STATUS BLOCK
HSAAKR    DS      1F     INSTRUCTION ADDRESS REGISTER
HSAIAR    DS      1F     ADDRESS KEY REGISTER
HSALSR    DS      1F     LEVEL STATUS REGISTER
HSAREGS   DS      8F     GENERAL REGISTERS 0 - 7
```

The contents of the various HSA locations (PSW,AKR,Etc,) will contain, at entry to your error exit routine, the values that were in the corresponding hardware registers at the time of the

exception. Upon entry to your error routine, general registers 1 and 2 will have been set to the SIA of your routine and to the address of your task's TCB, respectively.

Since entry to your error exit routine is made at the Event Driven Language level, the contents of the remaining general registers is dependent upon what instructions are executed.

What Happens When an Exception Occurs

If an exception (machine check, program check or soft exception trap) occurs during the execution of your task and you have specified a task error exit, as outlined above, the supervisor locates your TEECB. It then uses the TEEHSA pointer to locate your HSA and stores the hardware status information in it. Next, it retrieves the TEESIA pointer and sets it to zero to prevent recursive exceptions. Finally, it starts your task at the address it retrieved if that address is non-zero. If the TEESIA is zero or an exception occurs during any of this processing (if, for example, the TEECB is invalid), the supervisor treats the error as though no task Error Exit had been specified. Note that even if the TEESIA is zero, the supervisor still attempts to store the hardware status.

Since the supervisor zeroes TEESIA prior to starting your task, your error exit routine only gets control on the first exception that occurs, unless your routine restores TEESIA to its original condition. Zeroing TEESIA allows the supervisor to handle exceptions that occur in error exit routines, thus preventing recursion in the error handling process. When you implement a task error exit, do not restore TEESIA until the error exit routine has completed.

I/O ERROR LOGGING

The Event Driven Executive provides the capability to record device I/O errors into a log data set on disk or diskette and to display the log data set. The support is provided with a set of utilities and subroutines.

Recording the Errors

To activate I/O logging, the utility $LOG is loaded into any partition. The logging function can be deactivated, reactivated, and terminated after it becomes active.

```
NOTICE   PROGRAM   BEGIN
TERMX    IOCB      SCREEN=STATIC
NAMETAB  DATA      CL8'TERM1'
         DATA      CL8'TERM2'
         DATA      CL8'TERM3'
         DATA      CL8'TERM4'
BEGIN    MOVEA     #1,NAMETAB
         DO        4
         MOVE      TERMX,(0,#1),(8,BYTES)
         ENQT      TERMX
         PRINTEXT  'SYSTEM ACTIVE',LINE=0
         DEQT
         ADD       #1,8
         ENDDO
         PROGSTOP
         ENDPROG
         END
```

This example illustrates terminal access by using the name of the terminal. TERM1, TERM2, TERM3, and TERM4 must have been defined on a TERMINAL configuration statement. The use of the static screen mode insures that only physical line 0 of each screen will be altered. (LINE=0 for roll screens causes a page eject and erasure of information.)

Note: On a 4979 terminal, unprotected fields should be of even length.

Modifying the IOCB

Elements of the IOCB which may be modified by an application program are the terminal name, roll to static, and NHIST. The structure given here is provided for those special applications in which other elements may need to be modified; note that the structure may change with future versions of the Event Driven Executive.

BYTE(S)	ELEMENT	COMMENTS
0-7	Terminal Name	EBCDIC, blank filled
8	Flags	#CCBFLGS is described in the **Internal Design** manual under "Terminal I/O". Bit 0 off indicates that the name is the only element of the IOCB.
9	Top of working area	Equal to TOPM+NHIST
10	Top margin	TOPM or zero
11	Bottom margin	BOTM, or X'FF' if unspecified
12	Left margin	LEFTM or zero
13	Page size	Equal to X'00' if unspecified
14-15	Line size	Equal to X'7FFF' if unspecified
16	Current line	Initialized to TOPM+NHIST
17	Current indent	Left margin included
18-19	Buffer address	Zero if unspecified

Accessing a Static Screen

Line-oriented input/output instructions provide the most straightforward means for constructing and reading data from static screens. However, when individual data fields are accessed frequently, excessive screen flicker can result. This problem can be eliminated by transferring an entire screen image to the display device with one I/O operation. The following program will illustrate this procedure as well as some other important points relating to programming for static screens.

```
DISPLAY  PROGRAM  BEGIN
SCREEN   IOC      SCREEN=STATIC,BOTM=11,                C  2
                BUFFER=BUFF,RIGHTM=959
I        DATA     F'0'
BUFF     BUFFER   960,BYTES                               5
         DATA     X'0202'                                 6
NULLS    DATA     X'0000'                                 7
NUMS     DATA     48F'0'                                  8
VALS     TEXT     LENGTH=254                              9
BEGIN    ENQT     SCREEN                                 10
         ERASE    TYPE=ALL,LINE=0                        11
*
* THIS DO LOOP PLACES THE WORD "FIELD" AND THE VALUE
* OF "I" INTO THE TERMINAL BUFFER 96 TIMES.  THE
* ACTUAL CONTENTS OF THE TERMINAL BUFFER IS PRINTED
* WHEN THE "TERMCTRL DISPLAY" STATEMENT IS REACHED.
*
         DO       96,INDEX=I                             12
           PRINTEXT   'FIELD',PROTECT=YES                 13
           PUTEDIT    FORMAT1,VALS,((I)),PROTECT=YES      14
           PRINTEXT   ' ',PROTECT=YES                     15
           PRINTEXT  NULLS,PROTECT=YES                    16
         ENDO
         PRINTEXT  LINE=0                                 18
WRITE    PUTEDIT   FORMAT1,VALS,((NUMS,48)),           C 19
               ACTION=STG
         PRINTEXT  VALS,MODE=LINE,LINE=0                  21
         PRINTEXT  LINE=6,SPACES=8                        22
         TERMCTRL  DISPLAY                                23
         WAIT      KEY                                    24
         GOTO      (TRANSFER,QUIT),DISPLAY+2              25
TRANSFER READTEXT  VALS,MODE=LINE,LINE=6                  26
         GETEDIT   FORMAT1,VALS,((NUMS,48)),           C 27
               ACTION=STG
         ERASE     LINE=6,MODE=SCREEN,TYPE=DATA           29
         GOTO      WRITE                                  30
QUIT     DEQT                                             31
         PROGSTOP
FORMAT1  FORMAT (I2)
         ENDPROG
         END
```

This program accesses the top six lines of the screen in static mode and initially formats it with a sequence of protected fields. An array of integers is displayed on lines 0-5 and a pause is executed to allow the operator to enter a new set of values in corresponding positions of lines 6-11. The new values are then displayed on lines 0-5.

This program accesses the top six lines of a static screen and initially formats the screen with a sequence of protected fields. An array of integers is displayed on lines 0-5 of the screen and a pause is executed to allow the operator to enter a new set of values in corresponding positions of lines 6-11. The new values are then displayed on lines 0-5 of the screen.

The following numbers refer to lines (in the right margin) of the preceding example program.

2 Define the static screen with the terminal I/O buffer to be in the application program at BUFF, with a length of 960 bytes (half of the 4979 display screen).

5 Allocate storage for the buffer. Note that in this program the buffer is never accessed directly; the space is merely allocated here for use by the supervisor.

6 and 7 Define a TEXT message consisting of two null characters (EBCDIC code X'00').

8 and 9 Define the array of integers (initially zero) and the TEXT buffer which will be used for output of the data in EBCDIC form.

10 and 11 Acquire the terminal, erase all data and establish the screen position for the first I/O operation. Since several text strings will be concatenated to form the first output line, the screen position must be established in advance.

12 Begin a DO loop to construct the initial screen image. This will consist of 96 protected fields of the form FIELDxx, where xx is a sequential field number, each followed by one protected blank and two unprotected data positions. Note here the conditions required for forming a concatenated line; the protection mode of the PRINTEXT instructions must not change (either all PROTECT=YES or all PROTECT=No), and no intervening forms control operations can be executed.

13 Write 'FIELD' to the buffer.

14 Convert the sequence number to two EBCDIC characters and write it to the buffer.

15 Write a protected separation character.

16 Write the two null characters to define the data positions. Null characters will always generate unprotected positions on the screen; PROTECT=YES is nevertheless required here in order to maintain concatenation.

18 Write the concatenated line to the display. Any convenient line control operation or the DEQT instruction will accomplish this.

19 Convert the integer array to two-character EBCDIC values and store the resulting line in the TEXT buffer VALS.

21 Write the values into successive unprotected positions of the display beginning at LINE=0,SPACES=0. This "scatter write" mode is defined by MODE=LINE; without MODE=LINE the protected fields of the display would be overwritten.

22 Define the cursor to be at the first unprotected position.

23 Display the cursor at its defined position.

24 Wait for the operator to press an interrupt key.

25 Go to QUIT if PF1 was pressed. Go to TRANSFER if the ENTER key or any other key other than PF1 was pressed.

26 Read the updated values entered by the operator in lines 6-11. MODE=LINE indicates a "scatter read".

27 Convert the EBCDIC representations to binary and store the binary values in the arrary NUMS.

29 Erase the unprotected (data) fields in lines 6-11 of the screen.

30 Repeat

31 Release the terminal. The buffer designated in the IOCB will be released and the screen configuration restored to that defined by the TERMINAL statement.

In the previously described example program, the terminal I/O operations were all conviently performed through the concatenation of TEXT strings. If the application requires more complex formatting of the screen image, or if input of more than 254 bytes at a time is necessary, then direct access to the buffer is appropriate. See the PRINTEXT and READTEXT instructions in the Language Reference for details.

Formatted screen images can be created and saved in disk or diskette data sets with the utility program $IMAGE. The retrieval and display of such images can be simplified by employing a set of subroutines. An EXTRN statement must be coded for each subroutine name which is referenced, and AUTO=$AUTO,ASMLIB must be coded on the OUTPUT statement of the link-edit control data set.

In the calling formats given below, arguments which represent addresses to be passed to a subroutine must be enclosed within parentheses as shown. If the desired address is contained within a variable, then the name of that variable must be written without parentheses.

$IMOPEN Subroutine

This subroutine reads the designated image from disk or diskette into your buffer. You can also perform this operation by using DSOPEN or defining the data set at program load time, and issuing the disk READ instruction. Refer to the format description at the end of this section for data set size determination.

Syntax

```
label     CALL  $IMOPEN,(dsname,volume),(buffer),P2=,P3=

Required:  dsname,buffer
Defaults:  None
Indexable: None
```

Operands	Description
dsname	The address of a TEXT statement which contains the name of the data set. A volume label can be included, separated from the name by a comma.
buffer	The address of a BUFFER statement allocating the storage into which the image data will be read. The storage should be allocated in bytes, as in the following example:

End of Forms on 4973 and 4974 Printers

Terminal I/O goes into a wait state trying to print when one of the following situations occurs:

- The printer is in DISABLE (4973) or WAIT (4974) mode.
- The printer is out of paper and no terminal error exit is coded on your TASK/PROGRAM statement.
- The paper is jammed in the printer and no terminal error exit is coded on your TASK/PROGRAM statement.

You can correct this problem by doing the following:

- If in DISABLE or WAIT mode, set the switch to ENABLE (on 4973) or to PRINT (on 4974) to resume printing.
- If the printer is out of paper or the paper is jammed, set the mode switch to DISABLE or WAIT, add new paper or fix paper jam, and reset the mode switch to ENABLE or PRINT.
- If you have a terminal error exit coded on your TASK or program statements, you will get control at your error routine on all error conditions except DISABLE (4973) or WAIT (4974) modes.

Reading Modified Data on the 4978 Display

Both protected and unprotected fields on the 4978 are defined as a set of contiguous characters that may span line boundaries. A protected field ends when an unprotected field is encountered. Similarly, an unprotected field ends when a protected field is encountered. Neither an unprotected nor a protected field necessarily ends at an EDX partial screen boundary.

An unprotected field is considered a modified field when any character within the field is modified by the operator. The field may be read by the Event Driven Language READTEXT instruction with TYPE=MODDATA. Reading the field leaves its modified status unchanged. A modified field becomes an unmodified field by either writing or erasing all the characters in the field. For additional information, refer to IBM Series/1 4978-1 Display Station (RPQ D02055) and Attachment (RPQ D02038), General Information, GA34-1550.

CHAPTER 16. ADVANCED TOPICS

TRANSLATING COMPRESSED/NONCOMPRESSED BYTE STRINGS

The following two subroutines are used internally by $IMPROT and $IMDATA as well as by the utility program $IMAGE. They can also, however, be called directly by an application program and are described here because of their general utility.

The program preparation for applications calling $UNPACK and $PACK is similar to that when using the $IMAGE subroutines. That is, an EXTRN statement is required in the application and the aurocall to $AUTO,ASMLIB is required in the link- control data set (input to $LINK).

$UNPACK Subroutine

This subroutine moves a compressed byte string and translates it to noncompressed form.

Syntax

```
label     CALL  $UNPACK,source,dest,P2=,P3=

Required:  source,dest
Defaults:  None
Indexable: None
```

Operands	Description
source	The label of a fullword containing the address of a compressed byte string. (See Figure 43 on page 311 for the compressed format.) At completion of the operation, this parameter is incremented by the length of the compressed string.
dest	The label of a fullword containing the address at which the expanded string is to be placed. The length of the expanded string is placed in the byte preceding this location. The $UNPACK subroutine can, therefore, conveniently be used to move and expand a compressed byte string into a TEXT buffer.

The following example illustrates the use of the $UNPACK subroutine to unpack the compressed protected data of a $IMAGE screen format:

```
                 •
                 •
                 •
               MOVEA   #1,OUTAREA          POINT TO EXPAND BUFFER
               MOVEA   CPOINTER,CBUF+12    POINT TO FIRST BYTE OF
  *                                        COMPRESSED DATA
               MOVE    LINCNT,CBUF+4       INIT DO LOOP CTR
               MOVE    MOVELNG,CBUF+6      INIT MOVE LENGTH CODE
               DO      LINECNT
                 CALL  $UNPACK,CPOINTER,STRGPTR  UNPACK COMPRESSED
  *                                              DATA
                 MOVE  (0,BYTE),P3=MOVELNG  MOVE
  *                                UNPACKED DATA
                 ADD   #1,MOVELNG
               ENDDO
                 •
                 •
                 •
  OUTAREA      DATA    CL1920' '           WILL CONTAIN ALL OF THE
  *                                        UNPACKED DATA
  CPOINTER     DATA    A'0'                POINTER TO COMPRESSED DATA
  LINECNT      DATA    F'0'                NBR OF FORMAT LINES TO UNPACK
  STRGPTR      DATA    A(STRING)           ADDR OF TEMP LOCATION TO
  *                                        RECEIVE UNPACKED DATA
  STRING       TEXT    LENGTH=80           TEMP LOCATION TO RECEIVE
  *                                        UNPACKED DATA
  CBUF         BUFFER  1000,WORDS          CONTAINS $IMAGE FORMAT WITH
                                           WITH PACKED DATA
```

$PACK Subroutine

This subroutine moves a byte string and translates it to compressed form.

Syntax

```
label       CALL   $PACK,source,dest,P2=,P3=

Required:   source,dest
Defaults:   None
Indexable:  None
```

Operands	Description
source	The label of a fullword containing the address of the string to be compressed. The length of the string is taken from the byte preceding this location, and the string could, therefore, be the contents of the TEXT buffer.
dest	The label of a fullword containing the address at which the compressed string is to be stored. At completion of the operation, this parameter is incremented by the length of the compressed string.

$DISKUT3 Return Codes

The first word of the DSCB is posted with a return code to indicate whether the function was performed successfully (-1) or unsuccessfully. When a list of more than one function is specified, each function requested is processed in the sequence presented. A return code is posted for each function attempted.

Caution: If more than one function which references the same DSCB is requested on a single $DISKUT3 invocation, the return code set reflects only the results of the last function attempted using that DSCB.

The return codes set by $DISKUT3 and their meanings are as follows:

Code	Condition
1	Invalid request code (not 1-6)
2	Volume does not exist (All functions)
4	Insufficient space in library (ALLOCATE)
5	Insufficient space in directory (ALLOCATE)
6	Data set already exists - smaller than the requested allocation
7	Insufficient contiguous space (ALLOCATE)
8	Disallowed data set name, eg. $EDXVOL or $EDXLIB (All functions)
9	Data set not found (RENAME, RELEASE, OPEN)
10	New name pointer is zero (RENAME)
11	Disk is busy (ALLOCATE, DELETE, RENAME, RELEASE)
12	I/O error writing to disk (ALLOCATE, DELETE, RENAME, RELEASE)
13	I/O error reading from disk (All functions)
14	Data set name is all blanks (ALLOCATE, RENAME)
15	Invalid size specification (ALLOCATE)
16	Invalid size specification (RELEASE)
17	Mismatched data set type (OPEN, RENAME, DELETE, RELEASE)
18	Data set already exists - larger than the requested allocation
19	SETEOD only valid for data set of type 'data'
20	Load of $DISKUT3 failed ($RMU only)
21	Tape data sets not supported

Figure 44. $DISKUT3 return codes

Example: The following example illustrates the use of $DISKUT3. The use of all five functions (OPEN, ALLOCATE, RENAME, DELETE, and RELEASE) is shown.

```
TASK     PROGRAM  GO,DS=((DATA1,EDX002),(DATA2,EDX003))
         COPY     DSCBEQU
GO       EQU      *
         .
         .
         .
*  LOAD $DISKUT3 IN THE 'NON-OVERLAY' MODE, TO OPEN
*  DATA SET 'DATA3', ALLOCATE A NEW DATA SET 'DATA4', AND
*  RENAME AN EXISTING DATA SET 'DATA1'
*        LOAD     $DISKUT3,LISTPTR1,EVENT=$DISKUT3
         WAIT     $DISKUT3
         .
         .
         .
*  COMPUTE CURRENT SIZE OF THE DATA SET AND USE IT AS A
*  CALLING PARAMETER FOR A 'RELEASE' RECORDS CALL TO
*  $DISKUT3.
*  THE ASSUMPTION IN THIS PROGRAM IS THAT THE DATA
*  SET HAS BEEN WRITTEN SEQUENTIALLY.  THEREFORE,
*  '$DSCBNEXT' POINTS TO THE NEXT RECORD TO BE USED IN
*  THE DATA SET AND $DSCBNXT-1 IS THE NUMBER OF RECORDS
*  CURRENTLY IN USE.  WHENEVER THE FILE IS OPENED, $DSCBNXT
*  IS RESET TO X'0001'.  THE $DSCBNXT IS AUTOMATICALLY
*  INCREMENTED BY THE SYSTEM AS THE RECORDS ARE ACCESSED.
*
         SUBTRACT DSX+$DSCBNXT,1,RESULT=REQUEST5+4
         .
*  LOAD $DISKUT3, DELETE DATA SET 'DATA2'
*  AND RELEASE THE UNUSED SPACE IN 'DATA4'.
*
         LOAD  $DISKUT3,LISTPTR2,EVENT=$DISKUT3,PART=ANY
         WAIT  $DISKUT3
         .
         .
         PROGSTOP
         .
         .
$DISKUT3 ECB   0              SET INITIAL STATE TO ZERO
*
LISTPTR1 DC    A(LIST1)      POINTER TO LIST OF REQUEST
*                             BLOCK POINTERS
LISTPTR2 DC    A(LIST2)      POINTER TO ANOTHER LIST OF
*                             REQUEST BLOCK POINTERS
```

$DISKUT3 Use Example (Continued)

```
LIST1     DC    A(REQUEST1)
          DC    A(REQUEST2)
          DC    A(REQUEST3)
          DC    F'0'          END OF LIST FLAG
*
LIST2     DC    A(REQUEST4)
          DC    A(REQUEST5)
          DC    F'0'          END OF LIST FLAG
*
REQUEST1  DC    F'1'          REQUEST IS FOR AN 'OPEN'
          DC    A(DSY)        DSCB FOR 'DATA3'
          DC    F'0'          UNUSED FOR OPEN REQUESTS
          DC    F'-1'         FOR OPEN REQUESTS
*
REQUEST2  DC    F'2'          REQUEST IS FOR AN 'ALLOCATE'
          DC    A(DSX)        DSCB FOR 'DATA4'
          DC    F'50'         ALLOCATE 50 RECORDS
          DC    F'1'          DATA SET TYPE IS 'DATA'
*
REQUEST3  DC    F'3'          REQUEST IS FOR A 'RENAME'
          DC    A(DS1)        DSCB FOR 'DATA1'
          DC    A(NEWNAME)    ADDRESS OF NEW DATA SET NAME
          DC    F'-1'         FOR RENAME REQUESTS
*
REQUEST4  DC    F'4'          REQUEST IS TO 'DELETE'
          DC    A(DS2)        DSCB FOR 'DATA2'
          DC    F'0'          UNUSED FOR DELETE REQUESTS
          DC    F'-1'         FOR DELETE REQUESTS
*
REQUEST5  DC    F'5'          REQUEST IS TO 'RELEASE' SPACE
          DC    A(DSX)        DSCB FOR 'DATA4'
          DC    A(*-*)        NEW SIZE OF DATA SET
          DC    F'-1'         FOR RELEASE REQUESTS
*
          DSCB  DS#=DSY,DSNAME=DATA3
*
          DSCB  DS#=DSX,DSNAME=DATA4
          .
NEWNAME   DC    CL8'RENAMED'  NEW DATA SET NAME
          ENDPROG
          END
```

DSOPEN SUBROUTINE

DSOPEN is a subroutine that can be copied into your program. It opens a disk, diskette, or tape data set for input and/or output operations by initializing a DSCB. Only one DSCB can be open to a tape at a time. If a tape has been opened, a close must be issued before another open can be requested. The results of DSOPEN processing are identical to the implicit open performed by $L or LOAD for data sets specified in the PROGRAM statement.

Use DSOPEN to open a data set after the program has begun execution.

The following functions are performed:

- Verifies that the specified volume is online
- Verifies that the specified data set is in the volume
- Initializes the DSCB

Using DSOPEN adds 1056 bytes to the size of your program.

To use DSOPEN, you must first copy the source code into your program by coding:

```
        COPY   PROGEQU
        COPY   DDBEQU
        COPY   DSCBEQU
                 .
                 .
                 .
        COPY   DSOPEN
```

During execution, DSOPEN is invoked via the CALL instruction as follows:

```
        CALL   DSOPEN,(dscb)
```

Four optional parameters are also available. Three are error return addresses and the fourth is the address of an area in which DSOPEN saves a directory control entry (DCE) and the first directory member entry (DME).

The three error exit addresses are:

1. Data set not found

2. Invalid VOLSER

3. I/O error

Since the exit addresses and the area address lie within your program, they must be initialized by your program before it calls DSOPEN. DSOPEN automatically sets them to zero. The labels of these fields can be found in the beginning of the DSOPEN copy code. Since the four parameters are addresses within your program, you must insert (move) them to the beginning of the DSOPEN routine before calling it.

You must have a 256-byte work area labeled DISKBUFR in your program. The DSCB to be opened can be DS1-DS9 or a DSCB defined in your program via the DSCB statement. The DSCB must be initialized with a 6-character volume name in $DSCBVOL and an 8-character data set name in $DSCBNAM. To reopen a data set, the field $DSCBDDB in the DSCB must first be initialized to zero. Other fields are ignored. The volume name can be specified as 6 blanks, which causes the IPL volume to be searched for the data set.

After DSOPEN processing, #1 contains the number of the directory record containing the member entry and #2 contains the displacement within DISKBUFR to the member entry. The fields $DSCBR3 and $DSCBR4 contain the next available logical record data, if any, placed in the directory by SETEOD. Refer to the comments in the DSOPEN copy-code for additional details.

Only one dataset on any tape volume may be open at any one time. Multiple datasets, in a program header, or if opened by DSOPEN, cannot refer to more than one dataset per tape volume. If this is attempted, the second open attempt fails and takes the "INVALID VOLSER" error exit.

SETEOD

SETEOD is a copy code routine that updates the directory member entry (DME) of a disk directory to reflect the last record accessed up to the point in time SETEOD is invoked. Information on the DME can be found in Internal Design. The value in $DSCBNXT (relative record number to be used for next sequential READ or WRITE), minus one, is placed in the next available logical record field of the DME, so that it can be retrieved by subsequent calls of DSOPEN.

If the value of $DSCBNXT is 1 when SETEOD is performed, the DME is set to indicate that the data set is empty. Subsequent calls to DSOPEN cause $DSCBEOD to be set to X'FFFF', indicating that the data set is empty. If $DSCBEOD is zero, the length of the data set ($DSCBLEN) is used as the end-of-data (EOD) value.

SETEOD is used to indicate a logical end of file on disk. If your program does not SETEOD when creating or overwriting a file, the READ end of data exception will occur at either the physical end, or the logical end set by some previous use of the data set.

SETEOD can be used before, during or at the end of either input or output. It does not inhibit further I/O and can be used more than once. The only requirement is that the DSCB passed as input must have been previously opened.

The POINT function modifies the $DSCBNXT field. If SETEOD is used after a POINT, the relative record number pointed to, minus one, becomes the value placed in the directory by SETEOD.

SETEOD requires that the DSOPEN copy code, PROGEQU, and TCBEQU be included in your program. SETEOD uses the 256-byte DISKBUFR that is also used by DSOPEN. You invoke SETEOD as a subroutine through the Event Driven Language CALL statement, passing the DSCB and an I/O error exit routine pointer as parameters.

Using SETEOD adds 318 bytes to the size of your program.

To use SETEOD, you must first copy the source code into your program by coding:

```
COPY   PROGEQU
COPY   TCBEQU
COPY   DDBEQU
COPY   DSCBEQU
         .
         .
         .
COPY   DSOPEN
COPY   SETEOD
```

Calling Sequence

```
CALL   SETEOD,(DS1),(IOERROR)
```

where

DS1 Names a previously opened DSCB

IOERROR Names the routine in the application program to which control is passed if an I/O error occurs.

Reading End-of-Volume (EOV) Labels

The Event Driven Executive does not provide EOV processing. However, you may elect to add EOV processing to your application. To read a multi-volume data set the following steps can be used:

1. Vary the tape online (specifying the SL option).

2. Execute the program, reading and processing data records.

   When the end of the data set is reached, the END= exit routine of the READ statement will be entered. (If you do not use the END option, check for return code 10.)

3. Perform a CONTROL CLSOFF operation in the END= exit or when return code 10 is encountered.

   If the return code from the CONTROL operation is a +33 (EOV encountered), then the close processing has detected an EOV1 label. This means more data is contained on another reel. The CONTROL completes by rewinding the tape and setting it offline.

4. Issue a message (PRINTEXT) telling the operator to enter the volume serial number of the next tape.

5. Read (READTEXT) the volume serial number supplied by the operator from the terminal and place it in the $DSCBVOL field of the DSCB used to READ the data set.

6. Issue a message (PRINTEXT) telling the operator to place the next volume on an available tape drive and vary it online using $VARYON.

7. After the new tape has been varied online, call the DSOPEN subroutine to ready the data set for READ processing.

   Note: The new volume **must** be online ($VARYON) before DSOPEN is called.

8. Resume reading and processing as soon as the tape is opened

For a sample of the operator console sheet for the reading EOV process, see "Console Output for EOV Processing" on page 327
For a sample of a program to process an EOV condition while reading, see "Input EOV Processing Example" on page 329.

Support for:	Resident	Initialization
Program/Machine Check Log	250	
Relocating Loader		
With Addr Translator	4016	2352
Without Addr Translator	3068	2352
Floating Point Support		
Included	610	
Not Included	4	
Support of GETEDIT/PUTEDIT		
With Addr Translator	1602	
Without Addr Translator	1330	
Queue Processing Support	258	
$DEBUG Support	384	
Supervisor Patch Area	256	

Figure 50. (Part 3 of 3) V2.0 Supervisor Storage Requirements

Note: The transient program loader requires an area of 3840 bytes which will be overlaid by the loaded programs.

UTILITY PROGRAMS

The storage (in bytes rounded up to the next 256 byte increment) required by the Event Driven Executive utility programs:

$BSCTRCE	1792	
$BSCUT1	4864	
$BSCUT2	19712	
$COMPRES	3584	
$COPY	9216	
$COPYUT1	9984	
$DASDI	25600	
$DEBUG	6912	
$DICOMP	11264	
$DIINTR	9728	
$DISKUT1	7680	
$DISKUT2	9728	(+1280 if printing error log)
$DIUTIL	9216	
$DUMP	5888	
$EDIT1	9728	
$EDIT1N	11776	
$EDXASM	18944	(+5632 when assembling TERMINAL statements)
$EDXLIST	6144	
$FONT	5632	
$FSEDIT	22528	
$HCFUT1	2304	
$IAMUT1	13980	
$IMAGE	9728	
$INITDSK	6656	
$IOTEST	8960	
$JOBUTIL	5376	
$LINK	18688	
$LOG	5632	
$MOVEVOL	6144	
$PDS	1792	
$PFMAP	512	
$PREFIND	6144	
$PRT2780	2304	
$PRT3780	2560	
$RJE2780	9728	
$RJE3780	9984	
$TERMUT1	3072	
$TERMUT2	8192	
$TERMUT3	768	
$TRAP	5376	
$UPDATE	7936	
$UPDATEH	6400	
$VERIFY	21514	

SN34-0685-0