IBM /Technical Newsletter

This Newsletter No. SN34-0686
Date January 22, 1981
Base Publication No. SC34-0314-2
File No. S1-35

IBM Series/1
Event Driven Executive
Language Reference
Program Numbers: 5719-LM5 5719-LM6 5719-AM3 5719-XX2 5719-XX3 5719-MS1 5740-LM2 5719-LM3

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

3,4	86	218.1, 218.2 (added)
4.1, 4.2 (added)	89, 90	221 through 228
11, 12	93, 94	237, 238
27, 28	97, 98	243 through 246
33, 34	105 through 108	249, 250
34.1, 34.2 (added)	111, 112	251
37, 38	115, 116	251.1, 251.2 (added)
43, 44	119 through 122	252
44.1, 44.2 (added)	127	252.1, 252.2 (added)
45, 46	127.1, 127.2 (added)	253, 254
49, 50	128	259, 260
61, 62	149, 150	281, 282
65 through 68	163, 164	293, 294
68.1, 68.2 (added)	169 through 172	305, 306
69, 70	177 through 180	313, 314
79, 80	183, 184	317, 318
81	184.1, 184.2 (added)	335, 336
81.1, 81.2 (added)	185, 186	336.1, 336.2 (added)
82	193, 194	337 through 340
82.1, 82.2 (added)	197, 198	343, 344
83, 84	201, 202	351 through 354
84.1, 84.2 (added)	202.1, 202.2 (added)	413, 414
85	203 through 206	437, 438
85.1, 85.2 (added)	217, 218	

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.

gram.

A user application program has the following basic structure:

```
        PROGRAM
              other I/O definitions
                 .
                 .
              application program instructions
                 .
                 .
                 .
              application program data
                 .
                 .
                 .
        ENDPROG
        END
```

A complete source program starts with a PROGRAM statement and ends with the ENDPROG and END statements.

GENERAL INSTRUCTION FORMAT

Beginning with "Chapter 3. Instruction and Statement Descriptions" on page 51, each instruction is described in detail with brief remarks about the function, the syntax to be used to invoke a particular operation, the required parameters, and the defaults used if parameters are not specified. Each operand (or parameter) is listed and described.

Event Driven Language instructions have the following structure:

```
      label    operation     operands
```

The operands field in many cases has multiple entries, as indicated by the following example:

```
      label    op    opnd1,opnd2,..,opndn,P1=,P2=,...,Pn=
```

label	The label field, containing a symbolic label with a maximum of 8 characters. In most cases the label is optional. If used it must start in column 1.
operation	The operation field (or op) containing the instruction or statement.
operands	The operands field, containing the operands or parameters for the instruction. Maximum length is 254 characters.

P1=,P2=,Pn= The parameter-naming operands used to allow modification of the instruction parameters at execution time.

SYNTAX RULES

Syntactical coding rules are the same as those for the IBM Series/1 Macro Assembler. Some specific rules are as follows:

- An alphabetic string is 1 or more alphabetic characters (A - Z) or $, #, and @, the special characters.
- An alphameric string is 1 or more alphabetic characters or numeric characters (0 - 9).
- All upper case letters shown in the syntax descriptions starting in "Chapter 3. Instruction and Statement Descriptions" on page 51 must be coded as shown. This also applies to the comma immediately preceding the parameter and the equal sign (=) following. For example:

  ```
  ,PREC=
  ```

- Ellipses (...) indicate that a parameter may be repeated a variable (n) number of times.
- The vertical bar (|) between two operands indicates mutually exclusive operands; one or the other can be used but not both.
- All labels, instruction mnemonics, and parameter names must be alphameric strings of 1 to 8 characters in length, the first being alphabetic.
- Statement labels must begin in column 1. To continue a statement on another line, code a symbol in column 72, for example an asterisk (*), and begin the next line in column 16. Examples shown in this manual may not conform to the column spacing conventions due to limitations in the length of printed lines.
- Several instructions allow the use of immediate data or constants. These are called self-defining terms and improve the flexibility and ease of programming.
- Variable names, which are defined elsewhere by means of the EQU statement, must be coded with a leading plus sign (+) for proper compiler operation.

- Maximum number of delimeters for the operands field is 70. Delimiters are () or , or ' .
- The following labels are reserved for system use:

CONTROL BLOCK AND PARAMETER EQUATE TABLES

Application programmers sometimes wish to obtain data directly from system control blocks when coding specialized functions such as terminal commands (ATTNLIST exits), error exits (TASK ERRXIT or TERMERR) or a binary synchronous communication application. Many parameter lists and control blocks have equate tables which provide symbolic names for various values and the offset of each field relative to the beginning of the control block. Symbolic field names can be used in conjunction with index registers (see the "Address Indexing Feature" topic in this manual) to address the data in the control blocks. The symbolic values are often used as parameters.

These equate tables are:

BSCEQU	DSCBEQU	PROGEQU
CCBEQU	ERRORDEF	TCBEQU
CMDEQU	FCBEQU	TDBEQU
DDBEQU	IAMEQU	

Each equate table consists of a series of EQU statements which can be included in your program using the COPY statement. Although EQUs can be placed anywhere in a program, they are usually grouped together at either the beginning or the end. Some of the commonly used copy-code tables are briefly explained in the following sections. The control blocks themselves are described in _Internal Design_.

When compiling programs with the host or Series/1 Macro Assemblers, many equate tables are automatically included when a PROGRAM instruction is assembled. Tables included this way are PROGEQU, TCBEQU, DDBEQU, CMDEQU, and DSCBEQU.

BSCEQU

The BSCEQU equate table provides a map of the control block built by the BSCLINE system configuration statement.

BSCEQU is also the name of a macro in the macro libraries used with the host or Series/1 macro assembler. Do not attempt to COPY BSCEQU when using either macro assembler.

CCBEQU

The CCBEQU equate table provides a map of the control block (CCB) built by the TERMINAL system configuration statement.

CMDEQU

The CMDEQU equate table provides a map of the supervisor's emulator command table.

DDBEQU

The DDBEQU equate table provides a map of the device data block (DDB) built by the DISK system configuration statement.

DSCBEQU

The DSCBEQU equate table provides a map of the data set control block (DSCB) built by either the PROGRAM or DSCB statements.

ERRORDEF

The ERRORDEF equate table provides symbolic values for use in checking the return codes from the LOAD, READ, WRITE, and SBIO instructions.

FCBEQU

The FCBEQU equate table provides a map of an Indexed Access Method file control block (FCB) and its extension for use with the EXTRACT function.

IAMEQU

The IAMEQU equate table provides a set of symbolic parameter values for use in constructing parameter lists for CALLs to Indexed Access Method functions.

INDEXED ACCESS METHOD INSTRUCTIONS

DELETE	GET	PUTDE
DISCONN	GETSEQ	PUTUP
ENDSEQ	LOAD	PROCESS
EXTRACT	PUT	RELEASE

The Indexed Access Method is a data management system that operates under the IBM Series/1 Event Driven Executive. It provides callable interfaces to build and maintain indexed data sets and to access, by key or sequentially, the records in that data set. In an indexed data set, each of the records is identified by the contents of a predefined field called a key. The Indexed Access Method builds into the data set an index of keys that provides fast access to the records. Features of the Indexed Access Method include:

- Direct and sequential processing. Multiple levels of indexing are used for direct access; sequence chaining of data blocks is used for sequential access.

- Support for high insert and delete activity without significant performance degradation. Free space is distributed both throughout the data set and in a free pool at the end so that inserts can be made in place; space provided by deletes can be immediately reclaimed.

- Concurrent access to a single data set by several requesters. These requests can come from either the same or different programs. Data integrity is maintained by a file, block, and record level locking system that prevents access to that portion of the file that is being modified.

- Implementation as an independent task. A single copy of the Indexed Access Method serves and coordinates all requests. The buffer pool supports all requests and optimizes the space required for physical I/O; in the user program, the only buffer required is the one for the record currently being processed.

- An Indexed Access Method utility program which provides the capability to create, format, load, unload and reorganize an indexed data set.

- An Indexed Access Method utility program that verifies the status of an indexed data set and provides information concerning file structure and free space distribution.

The callable functions that comprise the Indexed Access Method are described in "Chapter 4. Indexed Access Method" on page 327 of this manual. They appear in alphabetic sequence by their function name, such as DELETE, DISCONN, and so on.

"Example 14: Use of Indexed Access Method" on page 414 is a complete program which illustrates many of the Indexed Access Method services. This example should help you understand the use of these services.

The Event Driven Executive Indexed Access Method Licensed Program (5719-AM3) is required to use these facilities.

LISTING CONTROL STATEMENTS

```
EJECT
PRINT
SPACE
TITLE
```

Listing control statements are used to identify program output listings, to provide blank lines in an assembly listing, and to designate how much detail is to be included in the listing. In no case are instructions or constants generated in the object program. With the exception of PRINT, listing control statements are not printed in the listing itself.

The format used to describe these instructions is the same as that used for describing the Event Driven Executive instruction set. However, they are part of the assembler facility itself and are not elements of the Event Driven Executive instruction set.

PROGRAM MODULE SECTIONING STATEMENTS

COPY
CSECT
ENTRY
EXTRN
WXTRN

The COPY statement allows you to copy into the your program a predefined source-program module from a data set.

The CSECT statement allows you to give names to the separately assembled modules of a program. These modules are then link-edited together to form a complete program.

The ENTRY, EXTRN, and WXTRN statements provide the information which allows the linkage editor ($LINK) to resolve symbolic address references among separately assembled program modules during link-edit processing.

Labels defined by CSECT and ENTRY statements, along with their addresses in the link-edited program are listed in the MAP portion of $LINK output.

PROGRAM SEQUENCING INSTRUCTIONS

DO	FIND
ELSE	FINDNOT
ENDIF	GOTO
ENDDO	IF

The IF, DO, and GOTO instructions provide the means for sequencing a program through the correct logic path based on the data and conditions generated during the execution of the program. IF and DO involve the use of relational statements which, based on a true or false condition, determine the next instruction to be executed. That next instruction must begin on a full-word boundary. Relational statements consist of a combination of data elements and are of the following:

EQ	-- Equal
NE	-- Not equal
GT	-- Greater than
LT	-- Less than
GE	-- Greater than or equal
LE	-- Less than or equal

The comparison is always arithmetic. A relational statement has the general format:

```
(data1,relcond,data2,width)
```

where:

width is optional,

relcond is one of the relational condition mnemonics,

data1 and data2 are data elements coded with the same syntax as other Event Driven Language instruction operands. Only data2 can contain immediate data. The immediate data can be decimal, hexadecimal, or EBCDIC data, must be an integer between -32768 and +32767, and will be converted to floating-point if necessary.

All hexadecimal constants must have a length that agrees with the width specification.

The default data width is 1 word (16 bits). The following table shows the allowed width specifications.

Specification	Data Element Width
BYTE	1 byte (8 bits)
WORD	1 word (16 bits) (integer)
DWORD	Doubleword (32 bits) (integer)
FLOAT	Single-precision floating-point (32 bits)
DFLOAT	Extended-precision floating-point (64 bits)
n	n bytes (relcond may only be EQ or NE)

QUEUE PROCESSING

DEFINEQ
FIRSTQ
LASTQ
NEXTQ

FIRSTQ, LASTQ, and NEXTQ provide the user with the capability to add entries to, or delete entries from a queue (defined by DEFINEQ) on a first-in-first-out or last-in-first-out basis. Entries are logically chained together and no associated data movement is required in the process. An entry is a 16-bit word which may, for example, be a data item, a record number in a data set, or the address of an associated data buffer. A queue is composed of a queue descriptor (QD) and one or more queue entries (QEs).

A QD is created by DEFINEQ and is 3 words in length. Word 1 is a pointer to the most recent entry on a chain of active QEs. Word 2 is a pointer to the oldest entry on a chain of active QEs. Word 3 is a pointer to the first QE on a chain of free QEs. If a queue is empty, words 1 and 2 contain the address of the queue (the address of the QD). If the queue is full, word 3 contains the address of the queue.

QEs are also created by DEFINEQ and are also 3 words in length. Word 1 is a pointer to the next most recent entry on a chain of active QEs. Word 1 of the most recent entry points to the QD. Word 2 is a pointer to the next oldest entry on a chain of active QEs. Word 2 of the oldest entry points to the QD. Word 3 of a free QE is a pointer to the next element in the free chain of QEs. Word 3 of the last QE in the free chain is a pointer to the QD. Word 3 of an active QE is the queue entry as described above.

Figure 2 on page 38 shows how a group of QEs are chained from a QD.

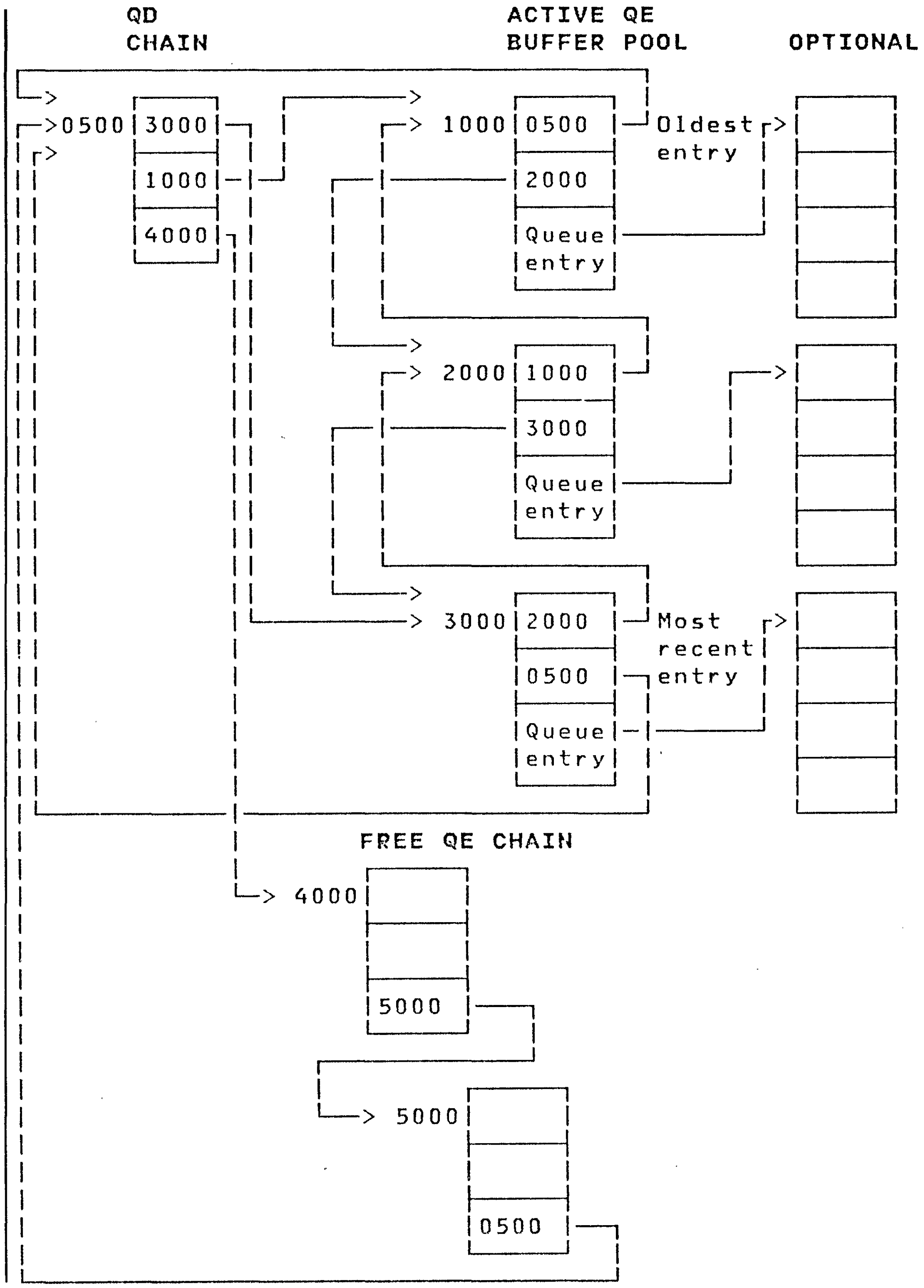


Figure 2. The Control Mechanism of Queue Processing

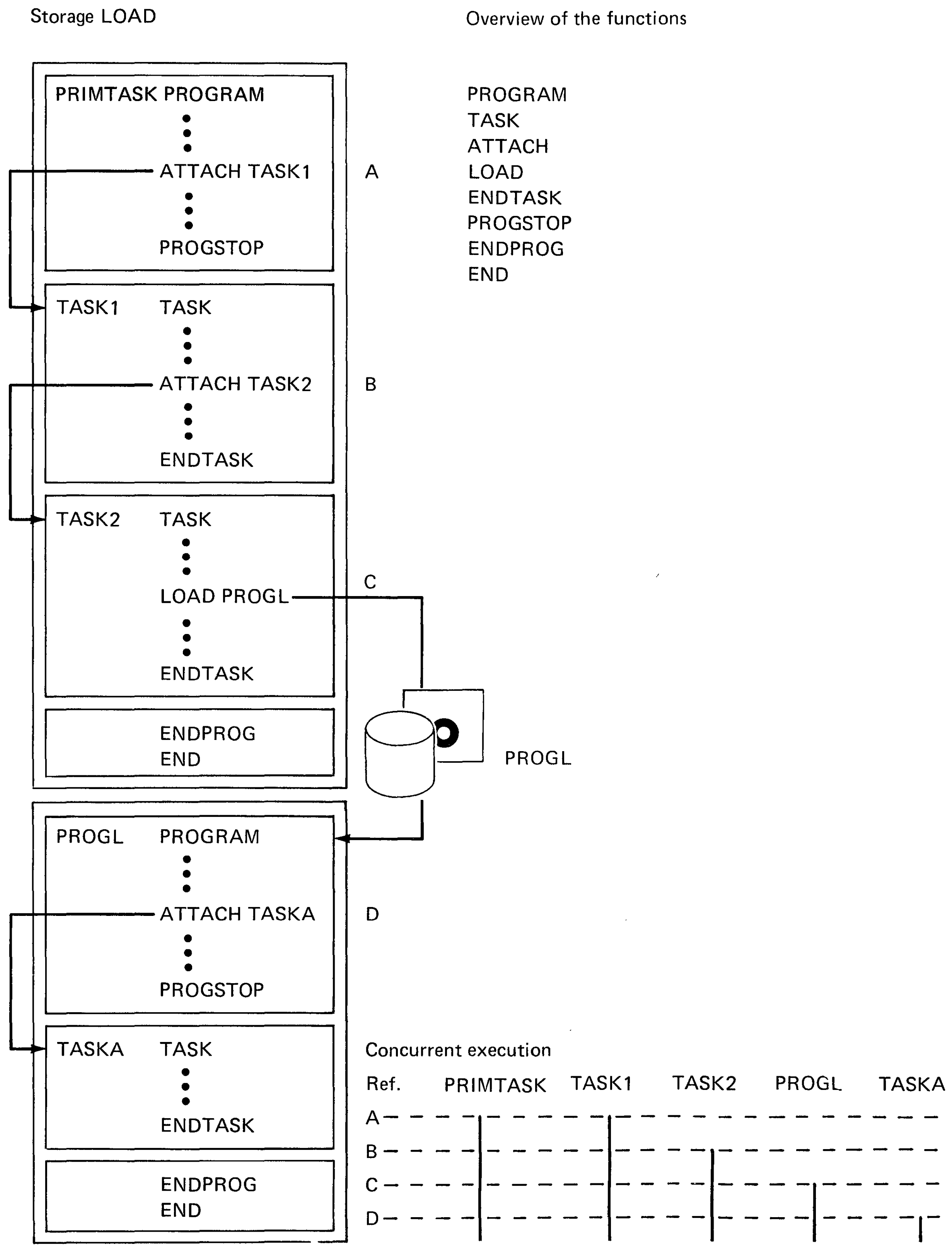


Figure 3. The Concurrent Execution of Multiple Tasks

TERMINAL I/O INSTRUCTIONS

DEQT	IOCB	READTEXT
ENQT	PRINTEXT	RDCURSOR
ERASE	PRINTIME	QUESTION
GETVALUE	PRINDATE	TERMCTRL
	PRINTNUM	

With few exceptions, you can write the terminal I/O instructions in an application program without concern for the type of terminal used or its hardware address. The terminal used by a program is assigned dynamically by the system as the one used to invoke the program and may vary from one invocation to the next without program change. Exceptions to this rule may exist with terminals which use special control characters or which have unique hardware capabilities such as graphics operations. Certain screen-oriented instructions are applicable only to the IBM 4978/4979 display.

The Event Driven Executive provides facilities to prevent conflicts among multiple programs using the same terminal. Each individual operation (read, write, or control) acquires exclusive control of the terminal for its duration. If you desire exclusive control for the duration of a sequence of instructions, for example to print a report, you can use the ENQT and DEQT instructions.

Error Handling

The application program can provide response to errors by means of the TERMERR oprand in the PROGRAM and TASK statements. In programs or tasks for which the TERMERR operand is coded with the label of an instruction, control is given to that instruction when an unrecoverable terminal I/O occurs. At that point the task code word, whose label is the task name, contains the error code, and the following word contains the address of the instruction during which the error occurred. If TERMERR is not coded, the error code is available in the task code word but program flow is not interrupted. Error codes are shown with the READTEXT and PRINTEXT instructions in this manual and in the Utilities, Operator Commands, Program Preparation, Messages and Codes. Use of TERMERR is the recommended method for detecting errors because the task code word is subject to modification by numerous system functions and may not always reflect the true status of the terminal I/O operations.

Because TERMERR receives control only when an actual I/O error occurs, it is important to note the way a PRINTEXT statement executes. A PRINTEXT statement does not result in immediate I/O operation or possible I/O error unless the TEXT statement contains an @ character or, the SKIP oprand is specified in a subsequent PRINTEXT statement. This information should be considered when coding a TERMERR routine.

Note that any I/O error that occurs during the execution of the PRINTNUM instruction does not cause control to be passed to TERMERR.

End of Forms: If you have coded the TERMERR operand on your PROGRAM or TASK statement, your error routine will get control on an end-of-forms condition or a paper jam condition. If you have not coded the TERMERR operand then on end-of-forms or a paper jam condition the terminal I/O task will enter a wait state.

TERMERR will not get control under the following two conditions: (a) a 4973 printer is switched to the Disable position, or (b) a 4974 printer is switched to the Wait position. These conditions cause the terminal I/O task to enter a wait state with no regard for TERMERR coding.

Data Representation

Output: Normally, alphameric text data to be written to a terminal is represented internally as a string of EBCDIC characters. The system translates the data to the code expected by the device. Means are also provided for writing untranslated data to the device for special purposes.

Integer numeric data is represented internally as binary integers of single-precision (2 byte) or double-precision (4 byte), or as floating-point numbers of single-precision (4 byte) or extended-precision (8 byte). You can specify translation to a designated external graphic form with numeric output instructions.

Input: Programs may request entry of text data in word mode without imbedded blanks. When several words are entered on a line, they must be separated from each other, and from any numeric entries on the same line, by one or more blanks. Programs such as the text-editor utility will also expect data entry in line mode, in which case the entire input line is stored internally as a string of EBCDIC characters. The ENTER key terminates an input operation in either word mode or line mode.

Integer numeric entries may be either decimal or hexadecimal, depending upon the program request. Decimal entries may include a plus (+) or minus (-) sign. When multiple numeric entries are made on the same line, the entries may be separated by blanks or by the delimiters comma (,) or slash (/). In conjunction with this rule, there are two means of indicating omitted values in a numeric sequence, namely the use of an asterisk (*) or two consecutive delimiters. Omitted values result in no change to the corresponding internal values, and their interpretation depends upon the utility or application program requesting the input. Allowable ranges for integer numeric input are given with the DATA instruction description in "Chapter 3. Instruction and Statement Descriptions" on page 51.

Forms Control

In order to achieve a high degree of device independence, all terminals, whether their display media be perforated paper, paper rolls, or electronic display screens, are treated according to line printer conventions. This means that within the limits imposed by differing page sizes and margins, the output from an application program will be identical in format for all terminal types. It is also possible to exercise direct control of forms movement by using the direct I/O capabilities of terminal I/O at the expense of device independence.

The forms control keyword parameters are common to several of the terminal I/O instructions. The values specified for any of the forms control parameters (SKIP, LINE, or SPACES) may be either constants or variables, and they may be indexed. Note that when forms parameters are specified on an I/O instruction, the forms operation always takes place before the data transfer.

Output Line Buffering: Two successive output instructions without the occurrence of the SKIP or LINE options, or the new line character @, result in concatenation of the data to form a single output line. The line is not displayed until a new line is indicated or the terminal is released through an explicit DEQT command, or the program terminates, or an input operation is performed. The default on TERMINAL and IOCB statements is OVFLINE=NO. When OVFLINE=NO is in effect and concatenated output exceeds the line-buffer capacity, subsequent output is lost until a new line indication is given or PRINTNUM issued. However, you can allow the generation of overflow lines by coding OVFLINE=YES.

Forms Interpretation for Electronic Display Screens: The PAGSIZE parameter for the IBM 4978/4979 Display is forced to 24. The margin settings TOPM,BOTM,LEFTM and RIGHTM delimit a logical screen which may be accessed independently of other logical screens. Once a logical screen has been defined and accessed, all I/O and forms control operations are defined relative to the margins of that screen. See the TERMCTL, ENQT, and IOCB statements in "Chapter 3. Instruction and Statement Descriptions" on page 51. Screen operations are described more fully under "Screen Management" on page 48.

Burst Output With Electronic Display Screens: Whenever the number of consecutive output lines reaches the logical screen size (BOTM-TOPM+1), the system will suspend further output, allowing the terminal operator to view the display. Upon operator signal (pressing the ENTER key on the 4978 or 4979), output continues until the screen is again filled or a pause for input occurs.

Prompting and Advance Input

As a terminal user, your interactive response with an application or utility program is generally conducted through prompting messages which request you to enter data. Once you have become familiar with the dialogue sequence, however, prompting becomes less necessary. The instructions READTEXT and GETVALUE include a conditional prompting option which enables entire screen image before data entry. Static screens are therefore distinguished from roll screens in the following ways:

- Forms control operations which would cause a page-eject for roll screens simply wrap around to the top for static screens. No automatic erasure is performed; selected portions of the screen are erased with the ERASE command.

- Protected fields may be written; this function is not available for roll screens.

- The cursor position, relative to the logical screen margins, may be sensed by the application program through the RDCURSOR command.

- Input operations directed to static screens normally do not cause a task suspension wait for the ENTER key; they are executed immediately. This allows the program to read selected fields from the screen after the entire display has been modified locally without program interaction by the operator. Operator/program signaling is provided through the program function keys and a special instruction, WAIT KEY.

- In order to allow convenient operator/program interaction to take place on a static screen, the QUESTION, READTEXT, and GETVALUE instructions are executed as if they were directed to a roll screen (automatic task suspension for input). READTEXT and GETVALUE are treated this way only when a prompt message is specified in the instruction.

- The character @ is treated as a normal data character. It does not indicate new line.

The utility program $IMAGE (see Utilities, Operator Commands, Program Preparation, Messages and Codes) can be used to construct formatted screen images in a user-interactive mode and save them in disk or diskette data sets. In addition, the images may be retrieved and displayed by application programs through the use of system provided subroutines. See "Formatted Screen Images", in the System Guide for details.

Operator Signals: An application program may wait at any point for a 4978/4979 terminal operator to press the ENTER key or one of the program function keys. This is done by issuing the WAIT KEY instruction.

When a key is pressed and the program operation resumes, the key is identified in the second task code word at taskname+2 (see "Attention Handling" on page 47). The code value for the ENTER key is 0, which is converted to a -1 by EDX. For the program function keys, the value is the integer corresponding to the the assigned function code; 1 for $PF1, 2 for $PF2, and so on.

The program function keys do not generate attention interrupts during execution of the WAIT KEY instruction. They only cause that instruction to terminate, allowing subsequent instructions to be executed.

TIMING INSTRUCTIONS

GETTIME
INTIME
PRINDATE
PRINTIME
STIMER

The timing functions are used in many different ways in the Event Driven language programs. The time-of-day clock can be displayed or it can be stored for data collection purposes. It can also be used to start and stop the execution of tasks.

Interval timers are also available for use by user programs and have a minimum time increment of 1 millisecond. The 4952 clock/comparator and the 4953/4955 timer feature #7840 are supported.

ATTNLIST

Task Control

The ATTNLIST statement provides entry to one or more user written asynchronous attention interrupt handling routines. When the attention key is pressed on a user terminal, the system will query the user for a 1-8 character command. By convention, commands beginning with $ are reserved for system use. All other character combinations are allowed.

The ATTNLIST statement produces a list of command names and associated routine entry points. Therefore, do not place the ATTNLIST statement between executable instructions. If the command entered matches one that is specified in the list, control passes to the associated user routine. However, if a user-routine entry point is not found, no action occurs, just as if the system query response had not matched any entry in the ATTNLIST.

The ATTNLIST allows you to control programs interactively from a terminal. These routines should be short because they are executed on hardware interrupt level 1 and may interfere with the execution of any other application programs. These routines must end with an ENDATTN instruction. Your program and the ATTNLIST routine execute asynchronously.

The next instruction to be executed will be the one following the instruction that was being executed when the ATTNLIST routine was entered.

The ATTNLIST statement can have only one list coded. The list can be up to 254 characters long and can contain a total of 24 ATTNLIST entries. A program may contain one LOCAL ATTNLIST statement and one GLOBAL ATTNLIST statement.

Coding of a LOCAL or a GLOBAL ATTNLIST causes a special ATTNLIST task control block (named $ATTASK) to be generated within your program. Routines invoked by ATTNLIST statements operate under the ATTNLIST task asynchronously with the other user or system tasks. System operator commands, however, operate as part of the system keyboard task within the supervisor. The following instructions are not recommended for use in an ATTNLIST routine: DETACH, ENDTASK, PROGSTOP, LOAD, STIMER, WAIT, TP, READ, WRITE, ENQT, and DEQT.

If the $DEBUG utility program is to be used to test your program, then the $DEBUG commands, listed in the Utilities, Operator Commands, Program Preparation, Messages and Codes cannot also be defined in an ATTNLIST in the program to be tested.

ATTNLIST

Syntax

```
label       ATTNLIST   (cc1,loc1,cc2,loc2,...,ccn,locn),
                       SCOPE=
SCOPE=

Required:   cc1,loc1
Defaults:   SCOPE=LOCAL
Indexable:  none
```

Operands	Description
cc1	The command identification requiring 1- to 8-alphameric characters. One exception is that $ is reserved for system use as a first character, except as noted under "Attention Handling" on page 47. The use of the 4979/4978 terminal program function keys to invoke ATTNLIST routines are defined there. Also see use of $DEBUG commands in Utilities, Operator Commands, Program Preparation, Messages and Codes.
loc1	Name of the routine to be invoked.
SCOPE=	An indicator of where the ATTNLIST is invoked from, either GLOBAL or LOCAL. GLOBAL allows the ATTNLIST command routines to be invoked from any terminal assigned to the same storage partition. LOCAL limits the invoking of the commands to the specific terminal (assigned to the same partition) from which the program containing the command was loaded. This is based on the premise that the partition assignment of the terminal has not been dynamically changed by a $CP command. A program may have one LOCAL ATTNLIST and one GLOBAL ATTNLIST.

Note: The following conditions apply to the ATTNLIST:

1. The $EDXASM compiler allows only one list with a maximum of 254 characters.

2. The Series/1 macro assembler and host assemblers allow multiple lists but with a maximum of 125 characters per list.

BUFFER

Data Definition

The BUFFER statement defines a data storage area. The standard buffer contains an index, a length, and a data buffer. The index may be used to indicate the current total number of words stored in the buffer. Both the index and the data buffer are initialized to 0.

Certain instructions, for example INTIME and SBIO, have an optional indexing facility wherein they can be used to add new entries sequentially to a buffer by implicitly referencing and incrementing the index word. The index can be thought of as a subscript to a one dimensional array. If a buffer becomes full and is to be reused, the index word must be reset to 0. Examination of the index word also indicates how many entries are currently in use in a buffer. You may assign a name to the index word in the BUFFER statement to provide for such program references.

BUFFER can be used to define the specialized storage area needed for use with the Host Communication Facility TP READ/WRITE instruction, and can also be used with the Terminal I/O instructions. Use of BUFFER for terminals is explained under the IOCB statement.

For a physical layout of a buffer see Figure 5 on page 67.

Syntax

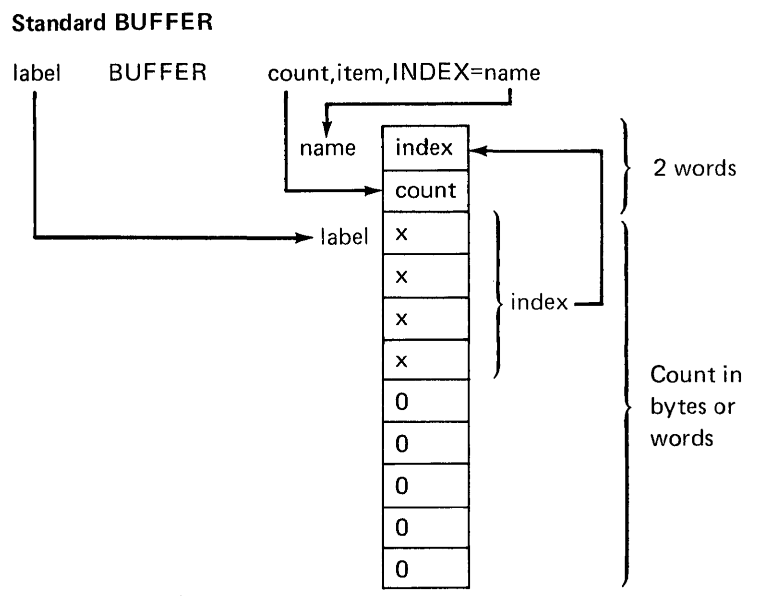
```
label       BUFFER    count,item,INDEX=

Required:   count
Defaults:   item=WORD
Indexable:  none
```

Operands	Description
count	The length of the buffer in terms of the item specified. In addition to the buffer itself, 2 words of control information are allocated. If the user program includes a READ instruction, the buffer area should be a multiple of 256 bytes.

item	Buffer type indicator. Code BYTE or BYTES if the buffer length is defined in terms of bytes. Code WORD or WORDS if the buffer length is defined in terms of words. The default for this operand is WORD. Code TPBSC to generate a buffer for use with the TP READ and WRITE statements (Host Communications Facility). BUFFER length must be specified in bytes if TPBSC is used.
INDEX=	A symbolic name assigned to the buffer index word. The parameter cannot be used if the item parameter is coded as TPBSC.

Note: Count and INDEX are maintained in terms of the number of data items (words or bytes) which the buffer can contain (total size) or currently contains, respectively. Index may also be regarded as the displacement of the next available location relative to the start of the buffer.



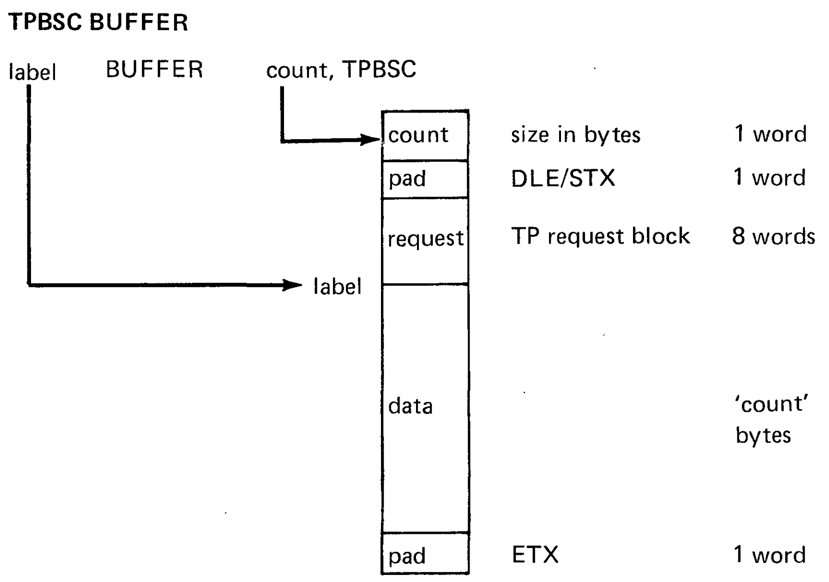


Figure 5. BUFFER Statement

CALL

Program Control

The CALL instruction executes a user-written or system subroutine. Up to five parameters may be passed as arguments to the subroutine. The first instruction of the subroutine is identified by a SUBROUT statement. If the called subroutine is a separate object module to be link-edited with your program, then you must also code an EXTRN statement for the subroutine name in the calling program.

Syntax

```
label        CALL     name,par1,...,par5,P1=,...,P6=

Required:    name
Defaults:    none
Indexable:   none
```

Operands	Description
name	The name of the subroutine to be executed.
parn	The parameters associated with the subroutine. The following are passed to the subroutine:

- up to five, explicit, single precision, integer constants

or

- symbolic label of single precision, integer constants

or

- null parameters

The actual constant or the value at the named location is moved to the corresponding subroutine parameter. Updated values of these parameters are returned by the subroutine.

If the parameter name is enclosed in parentheses, for example, (par1), the address of the variable is passed to the subroutine parameter. Such an address may be the label of the first word of any type of data item or data array. Within the subroutine it will be necessary to move the passed address of the data item into one of the index registers, #1 or #2, in order to refer to the actual data item location in the calling program.

If the parameter name enclosed in parentheses is a symbol defined by an EQU statement, the value of the symbol is passed as the parameter.

If the parameter to be passed is the value of a symbol defined by an EQU statement, it can also be preceded by a plus (+) sign. This causes the value of the EQU to be passed to the subroutine. If not preceded by a +, the EQU is assumed to represent an address and the data at that address is passed as the parameter.

Px= Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
        CALL    PROG,5,        The value 5 and the null parameter
                               0 is passed to PROG

        CALL    SUBROUT,PARM1,(PARM2),+FIVE

                                The parameters passed to SUBROUT
                                are the contents of PARM1, the
                                address of PARM2 and the value
                                of the EQU symbol FIVE
```

Figure 6 shows the control flow when using a CALL statement.

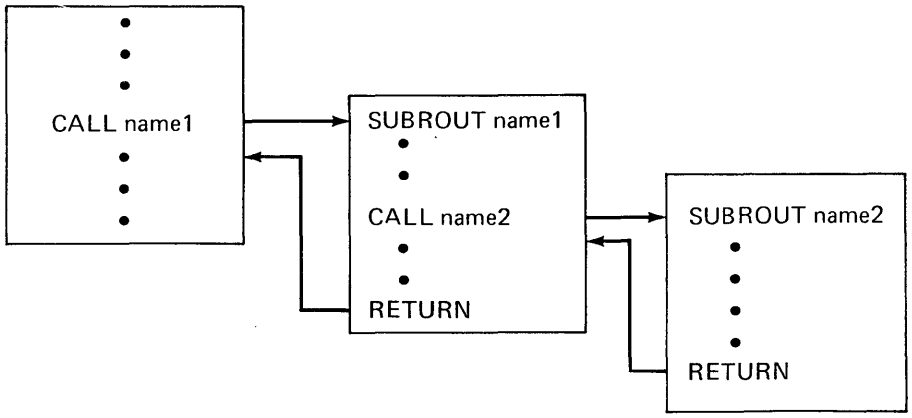


Figure 6. Execution of Subroutines

CALLFORT

Program Control

The CALLFORT instruction calls a FORTRAN program or subroutine from an Event Driven Executive program. If a FORTRAN main program is called, the name you specify on the name parameter is the name coded in the FORTRAN PROGRAM statement or the default name MAIN if no PROGRAM statement was coded. If a FORTRAN subroutine is called, specify the subroutine name. Parameters may be passed to FORTRAN subroutines. Standard FORTRAN subroutine conventions apply to the use of CALLFORT.

For a more complete description of the use of the CALLFORT statement, see the IBM Series/1 FORTRAN IV Licensed Program 5719-F01, F03, User's Guide, SC34-0134.

Syntax

```
label       CALLFORT  name,(a1,a2,...,an),P=(p1,p2,..pn)

Required:   name
Defaults:   none
Indexable:  none
```

Operands	Description
name	The name of a FORTRAN program which consists of 1 to 6 alphabetic or numeric characters, the first of which must be alphabetic. This name, or entry point, must also be coded in an EXTRN statement.
a	Each a is an actual argument that is being supplied to the subroutine. The argument may be a constant, a variable, or the name of a buffer.
P=	Parameter naming operands (See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions). A list of names of up to 8 characters each can be provided. These names are assigned to the parameter list entries for the arguments specified in the a operand in the order specified.

CONVTB

Data Formatting

The CONVTB instruction converts a binary value to an EBCDIC string. Both integer and floating-point formats are provided. In addition, both the normal floating-point notation and E notation are provided.

Syntax

```
label       CONVTB   opnd1,opnd2,PREC=,FORMAT=,P1=,P2=

Required:   opnd1,opnd2
Defaults:   PREC=S,FORMAT=(6,0,I)
Indexable:  opnd1,opnd2
```

Operands	Description
opnd1	The name of an area in storage where the converted results will be placed. The address must be the leftmost byte of the area. The converted results will be in EBCDIC.
opnd2	The name of the variable to be converted to EBCDIC. You must know the format of the data. The following opnd2 types are supported:

Single-precision integer	-- 1 word
Double-precision integer	-- 2 words
Single-precision floating-point	-- 2 words
Extended-precision floating-point	-- 4 words

PREC= The PREC keyword is used to specify the form of opnd2. The allowable values are:

S - Single-precision integer
D - Double-precision integer
F - Single-precision floating-point
L - Extended-precision floating-point

FORMAT=(W,D,T) The format of the value converted.

W = Field width in bytes of EBCDIC field

D = Number of digits to the right of decimal point. Valid for floating-point variables only. For integer values, code a 0 here.

T = Type of EBCDIC Data as follows:

I- Integer XXXX

F- Real number XXXX.XXX

E- Real number of exponent (E) notation

This notation uses the form:

SX.XXESYY

where:

S = Optional sign character (+ or -), default = (+)
X = Characteristic, 1 to 6+ digits (for PREC=E, or 15 digits for PREC=L.)
Note: Some but not all 7 digit characteristics can be represented by a 4 byte, floating-point, binary number.
. = Decimal point anyplace within characteristic
E = Designation of E notation
YY = Mantissa, range -85 to +75. The base is 10.

Px= Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Following are the return codes returned at taskname (See PROGRAM/TASK statements).

Return Codes

Code	Description
-1	Successful completion
3	Conversion error

Operation: The Convert Binary to EBCDIC instruction accepts both integer and floating-point variables and converts them into an EBCDIC character string. The format of the EBCDIC character string is defined by the use of the operands PREC and FORMAT. The following examples should help define the capabilities of this instruction.

Examples:

This example demonstrates a use of the CONVTB instruction.

```
HEADER    EQU      *
          READTEXT TITLE,TITLEMSG
          PRINTEXT SKIP=4
*
CONVERT   EQU      *
          CONVTB   ENUMEXP,BNUMEXP
          PRINTEXT '@NUMBER OF EXPERIMENTS CONDUCTED : ', X
                SKIP=1
          PRINTEXT ENUMEXP
*
          CONVTB   EMANHRS,BMANHRS,PREC=F,FORMAT=(10,2,F)
          PRINTEXT '@TOTAL MANHOURS EXPENDED              X
                ON PROJECT : ',SKIP=1
*
          PRINTEXT EMANHRS
*
          CONVTB   EAVERAGE,BAVERAGE,PREC=L,              X
                FORMAT=(20,14,E)
*
          PRINTEXT '@AVERAGE PENETRATION IN CONCRETE      X
                (MILLIMETERS) : '
*
          PRINTEXT EAVERAGE
                 •
                 •
                 •
BNUMEXP   DATA    F'0'        BINARY VALUE - # EXPERIMENTS
ENUMEXP   TEXT    LENGTH=6    EBCDIC VALUE - # EXPERIMENTS
BMANHRS   DATA    D'0'        BINARY VALUE - MAN-HOURS USED
EMANHRS   TEXT    LENGTH=8    EBCDIC VALUE - MAN-HOURS USED
BAVERAGE  DATA    D'0'        BINARY VALUE - AVERAGE RESULT
EAVERAGE  TEXT    LENGTH=20   EBCDIC VALUE - AVERAGE RESULT
TITLE     TEXT    LENGTH=40
TITLEMSG  TEXT    'ENTER A 40 CHARACTER TITLE FOR YOUR    X
                  REPORTS'
```

If the initial value of BNUMEXP is X'0038', the BMANHRS value is X'431B0C00', and BAVERAGE is X'4087915E8CA84482', the above section of code will produce the following lines of output:

```
         FINAL STATISTICS FOR THIS PROJECT FOLLOW : '
```

```
NUMBER OF EXPERIMENTS CONDUCTED : 56

TOTAL MAN-HOURS EXPENDED ON PROJECT : 432.75

AVERAGE PENETRATION IN CONCRETE
(MILLIMETERS) : .52956191000000E+00
```

The following are prototype statements of the CONVTB instruction.

Integer Example

```
          CONVTB TEXTA,VALUE,PREC=S,FORMAT=(8,0,I)
            .
            .
            .
 VALUE    DATA   F'12345'
 TEXTA    TEXT   LENGTH=8
```

The value 12345 in the variable VALUE will be converted to EBCDIC at TEXTA in the following format:

```
 bbb12345
```

If conversion of double-precision integers is required, then PREC=D is coded.

Floating-Point Example

```
          CONVTB  TEXTB,VALUE,PREC=F,FORMAT=(15,4,F)
          CONVTB  TEXT1,VALUE1,PREC=L,FORMAT=(20,14,E)

 VALUE    DATA    E'62421.16'
 VALUE1   DATA    L'4926139.2916'
 TEXTB    TEXT    LENGTH=15
 TEXT1    TEXT    LENGTH=20
```

The following EBCDIC character strings would result (b represents blanks):

```
 TEXTB=bbbbb62421.1600

 TEXT1=b.49261392916000Eb07
```

Remember that the conversion routines assume that the type of variable to be converted is as specified by the PREC operand. If the internal format of the variable is something other than specified by the PREC operand, incorrect results will occur.

CONVTD

The CONVTD instruction converts an EBCDIC character string to a binary arithmetic value. Both integer and floating-point variables are allowed.

Syntax

```
label       CONVTD    opnd1,opnd2,PREC=,FORMAT=,P1=,P2=

Required:   opnd1,opnd2
Defaults:   PREC=S,FORMAT=(6,0,I)
Indexable:  opnd1,opnd2
```

Operands	Description
opnd1	The name of a variable where the result of the conversion is to be stored. You must insure that enough space is reserved to accommodate the results.

Single-precision integer	-- 1 Word
Double-precision integer	-- 2 Words
Single-precision floating-point	-- 2 Words
Extended-precision floating-point	-- 4 Words

opnd2 The address of the first character of the EBCDIC character string.

Allowable ranges for data values are:

Single-precision integer	-32768 to 32767
Double-precision integer	-2147483648 to 2147483647
Single-precision floating-point	6+ decimal digits*
Extended-precision floating-point	15 decimal digits*

Note: Some but not all 7 digit characteristics can be represented by a 4 byte, floating-point, binary number.

*Exponent range is from 10 to the -85th through 10 to the 75th.

PREC= The form of opndl.

S	Indicates single-precision integer
D	Indicates double-precision integer
F	Indicates single-precision floating-point
L	Indicates extended-precision floating-point

FORMAT=(W,D,T) The format of the value being converted.

W = Field width in bytes of EDCDIC field

D = Number of implied decimal positions if no decimal point is in input (valid for floating point only). For integer values code a 0.

T = Type of EBCDIC data as follows:

I	Integer	xxxxx
F	Real number	xxx.xx
E	Real number in E notation (see CONVTB for a description of E notation)	

Px= Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Following are the return codes returned at taskname (See PROGRAM/TASK statements).

Return Codes

Code	Description
-1	Successful completion
1	Invalid data encountered during conversion
2	Field omitted
3	Conversion error

Operation: The Convert EBCDIC to Binary instruction accepts a variety of input formats. The following examples will help to define the various types accepted.

Integer Example

```
            CONVTD   VALUE,TEXT,PREC=S,FORMAT=(8,0,I)

VALUE       DATA     F'0'
TEXT        TEXT     '12345',LENGTH=8
```

The value in EBCDIC, 12345, will be converted to a single precision binary value and stored at VALUE as X'3039'. Double-precision integers can also be converted by using the PREC=D parameter and using a 2 word variable at VALUE.

Floating-Point Example

```
            CONVTD   VALUE,TEXT1,PREC=F,FORMAT=(10,2,F)
            CONVTD   VALUE1,TEXT2,PREC=L,FORMAT=(15,0,E)

VALUE       DATA     2F'0'
VALUE1      DATA     4F'0'
TEXT1       TEXT     '100.5',LENGTH=10
TEXT2       TEXT     '0.1005E3',LENGTH=15
```

Both values shown in the TEXT statements result in the same binary data values being stored in the two DATA statements. The only difference is that at VALUE1 an extended-precision value is stored.

The following example demonstrates a use of the CONVTD instruction:

```
CONVERT    EQU      *
           READTEXT UNIT,@ENTER UNIT NUMBER  '
           CONVTD   BUNIT,UNIT,PREC=S,FORMAT=(6,0,I)
*
           READTEXT MILES,'@ENTER MILES FROM FIRE '
           CONVTD   BMILES,MILES,PREC=F,FORMAT=(10,4,F)
*
           READTEXT RESPONSE,'@ENTER UNIT RESPONSE TIME '
           CONVTD   BRESPONS,RESPONSE,PREC=L,FORMAT=(15,8,E)
*           •
*           •
*           •
UNIT       TEXT     LENGTH=6     EBCDIC VALUE/UNIT I.D.
BUNIT      DATA     F'0'         BINARY VALUE/UNIT I.D.
MILES      TEXT     LENGTH =10   EBCDIC VALUE/MILES FROM FIRE
BMILES     DATA     D'0'         BINARY VALUE/MILES FROM FIRE
RESPONSE   TEXT     LENGTH=15    EBCDIC VALUE/RESPONSE TIME
BRESPONS   DATA     2D'0'        BINARY VALUE/RESPONSE TIME
```

Assuming that unit # 6553 took 42.45292378 minutes to respond to an alarm for a fire 41.5429 miles from the station, the values would be:

	EBCDIC	HEXADECIMAL
UNIT	6553bb	X'F6F5F5F34040'
BUNIT	N/A	X'1999'
MILES	41.5429bbb	X'F4F14BF5F4F2F9404040'
BMILES	N/A	X'42298AFB'
RESPONSE	42.45292378bbbb	X'F4F24BF4F5F2F9F2F3F7F840404040'
BRESPONS	N/A	X'422A73F2D016AE42'

The EBCDIC field should contain only those characters that are valid for the operation being performed. For example:

- Integers

  Leading blanks
  Sign character + or -
  Digits 0 through 9
  Trailing blanks

- Floating-point

  Leading blanks
  Sign character + or -
  Digits 0 through 9
  Decimal-point
  The character E, if E notation, followed by a sign character, + or -, or the digits 0 through 9.

The following deal with integer and floating-point characters as previously defined.

- A comma (,) or slash (/) is treated as a delimiter. The remainder of the field is not scanned

  If the data was found preceding the delimiter, the value is returned in the target field and "successful completion" (-1) is indicated

  If no data was found preceding the delimiter, the target field is unchanged and "field omitted" (2) is indicated

- · An asterisk (*) or period (.) in an integer field:

  is treated as an invalid character if preceded by data. The value is returned to the target field and "invalid data occured during conversion" (1) is indicated

  is treated as a delimiter if not preceded by data

  The target field is unchanged and "field omitted" (2) is indicated

- Any other character (like an alphabetic character) is treated as invalid, including characters that follow trailing blanks

  If data was found preceding the invalid character, the value is returned in the target field and "invalid data occured during conversion" (1) is indicated

  If no data was found preceding the invalid character, the target field is unchanged and "conversion error" (3) is indicated

If the field contains only blanks, or only blanks and a sign, the value zero is returned in the target field and "successful completion" (-1) is indicated.

If the value is outside the allowable range, the value of the target field is unknown and "conversion error" (3) is indicated.

If the field contains a decimal point, the target field remains unchanged and "field omitted" (2) is indicated. The decimal point in a floating-point field must be preceded by blanks and followed by a blank. A sign preceding the decimal point yields a target value of zero and "successful completion" (-1) is indicated.

For example, using the default format (6,0,I) produces the results shown by the following table:

INPUT	RETURN CODE	OUTPUT
12	-1	12
12,	-1	12
12/	-1	12
	-1	0
12C	1	12
12.B	1	12
12 C	1	12
,	2	X (target field unchanged)
/	2	X (target field unchanged)
*	2	X (target field unchanged)
.	2	X (target field unchanged)
A	3	X (target field unchanged)
1234567	3	?? (value of target field unknown)

COPY

Program Module Sectioning

The COPY instruction copies a predefined source program module into your program. The module to be copied must exist in a disk or diskette data set. The specified source statements are copied immediately following the COPY statement. The program module to be copied must not contain a COPY statement.

Syntax

```
blank        COPY     symbol

Required:    symbol
Defaults:    none
Indexable:   none
```

Operands	Description
symbol	The symbolic name of the source module on disk or diskette that is to be copied into your program.

- The assembler program $EDXASM provides a restricted implementation of the COPY statement. The names of the volumes which may contain modules which may be referenced must be in the control list $EDXL. See the description of $EDXASM in the Utilities, Operator Commands, Program Preparation, Messages and Codes for details on how you can add your own '*COPYCOD' definitions to those supplied as standard definitions in $EDXL.

- The Series/1 macro assembler provides a full implementation of the COPY statement as part of the Event Driven Executive Macro Library (5719-LM5 or 5719-LM6). See the IBM Series/1 Event Driven Executive Macro Assembler (5719-ASA) for details on using this COPY statement.

- The System/370 macro assembler also provides a full implementation of the COPY statement as part of the IBM System/370 Program Preparation Facility FDP (5798-NNQ). See the IBM System/370 Program Preparation Facility, SB30-1072 for details on using this COPY statement.

Valid codes for type are:

Code	Type Constant	Storage Format
C	EBCDIC	8-bit code for each character
X	Hexadecimal	4-bit code for each digit
B	Binary	1-bit for each digit (not allowed with $EDXASM)
F	Fixed-point	Signed, fixed-point binary; 2 bytes
H	Fixed-point	Signed, fixed-point binary; 1 byte
D	Fixed-point	Signed, fixed-point binary; 4 bytes
E	Floating-point	Floating-point binary; 4 bytes
L	Floating-point	Floating-point binary; 8 bytes
A	Address	Value of address or expression; 2 bytes

Allowable ranges for data values are:

Single-precision integer	-32768 to 32767
Double-precision integer	-2147483648 to 2147483647
Single-precision floating-point	6+ decimal digits*
Extended-precision floating-point	15 decimal digits*

*Exponent range is from 10 to the -85th to 10 to the 75th

Floating point constants can be expressed as real numbers with decimal points, for example 1.234, or can be expressed in exponent (E) notation. E notation uses the form:

SX.XXESYY

where:

S = Optional sign character (+ or -); default = (+)
X = Characteristic 1 to 6+ digits (for PREC=E, or 15 digits for PREC=L)
Note: Some but not all 7 digit characteristics can be represented in a 4 byte, floating-point, binary number.
. = Decimal point anyplace within characteristic
E = Designation of E notation
YY = Mantissa, range -85 to +75. The base is 10.
(for example, 3.1415E-2 = .031415)

Character constants (C) can include an explicit length specification for the field by specifying the type as CLn where n is the length of the field. If the value operand is smaller than the field length, the balance of the field is filled with blanks.

Example

```
BINCON   DATA   B'001100001111'   Hexadecimal 30F in
                                  binary

A        DATA   F'1'              Decimal constant 1

BUF      DC     128F'0'           128 words of 0

CHAR     DATA   C'XYZ'            EBCDIC String 'XYZ'

BLANK    DC     80C' '            80 EBCDIC blanks

C8       DC     CL8'$'            $ followed by 7 blanks

HEXV     DATA   X'00F1'           Decimal 241 in
                                  hexadecimal

ADDR     DATA   A(BUF)            Address of 'BUF'

DBL      DATA   D'100000'         2-word decimal constant
                                  100,000

F1       DATA   E'1.234'          Floating-point value 1.234

F2       DATA   4E'0.123'         Four Floating-point values of
                                  0.123 (4 bytes each value)

L2       DATA   4L'12345678.9'    Four Extended-precision
                                  Floating-point values of
                                  12345678.9 (8 bytes each
                                  value

L3       DATA   L'.123456E-40'    Extended-precision float-
                                  ing point in exponent form

MANY     DATA   F'1',D'2'         A word of 1 and a double
                                  word of 2
```

Example:

```
 WR1DCB   DCB  SE=YES,DVPARM1=0300,DVPARM2=3048,            C
               DVPARM3=1100,DVPARM4=RESTAT,                 C
               CHAINAD=WR2DCB,COUNT=120,DATADDR=MSG1

|WR2DCB   DCB  SE=YES,DVPARM1=20A0,DEVMOD=6F,               C
               DVPARM4=RESTAT

 MSG1     DATA 120X'00'
 RESTAT   DATA 2F'0'
```

DEFINEQ

Queue Processing

The DEFINEQ statement defines the queue descriptor (QD) and the set of queue elements (QEs) used by FIRSTQ, LASTQ, and NEXTQ. DEFINEQ can optionally define a pool of data storage areas or data buffers. For additional information refer to the discussion of queue processing in Chapter 2 of this manual.

Syntax

```
label        DEFINEQ  COUNT=,SIZE=

Required:    label, COUNT=
Defaults:    none
Indexable:   none
```

Operands	Description
COUNT=	The number of 3-word queue elements to be generated. An additional 3-word QD will be generated and the first word will be assigned the name specified in the label on the DEFINEQ statement. The COUNT operand must be specified using a self-defining term. An equated value is not allowed. This operand must also be a positive number greater than 0.
SIZE=	The size, in bytes, of each buffer (data area) to be included in the buffer pool in the initial queue. As many such buffers will be generated as specified in the COUNT operand. Each such buffer is initialized to binary zeros. Each QE in the queue will contain the address of an associated buffer in the buffer pool. If the SIZE operand is not specified, all QEs will be generated to be in the free chain and the queue will be defined as empty. If SIZE is specified, all QEs will be included in the active chain and the queue will be defined as full.

Example: See the example following the NEXTQ instruction.

DEQT

Terminal I/O

The DEQT statement releases the terminal which was previously acquired with an ENQT instruction. A task may issue successive ENQTs directed to the same terminal before issuing a DEQT. Until DEQT is executed, however, ENQTs directed to other terminals are ignored. If a terminal configuration was established by ENQT, then DEQT restores the configuration to that defined by the TERMINAL system configuration statement. DEQT also forces partially full buffers to be written to the terminal and completes all pending I/O.

Syntax

```
label       DEQT

Required:   none
Defaults:   none
Indexable:  none
```

Operands	Description
none	none

Example of ENQT and DEQT

```
          ENQT  $SYSPRTR
           .
           .
          DEQT
          ENQT  TERM1,BUSY=ALTERN
           .
           .
          DEQT
           .
           .
ALTERN    ENQT  $SYSLOG
           .
           .
TERM1     IOCB  TTY1,PAGSIZE=24
```

DETACH

Task Control

The DETACH instruction removes a task from operational status. A task may only detach itself. If a task is reattached, execution proceeds with the next instruction after the DETACH in the reattached task.

Syntax

```
label       DETACH    code,P1=

Required:   none
Defaults:   code = -1
Indexable:  none
```

Operands	Description
code	The posting code to be inserted in the terminating ECB ($TCBEEC) of the task being detached. Additional information on this statement can be found in the Internal Design manual (LY34-0168).
P1=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

DSCB

Disk/Tape I/O

The DSCB statement generates a data set control block (DSCB). A DSCB provides the information required to access a data set within a particular volume. One DSCB is generated in the program header for each data set specified in the DS parameter of the PROGRAM statement. The name of each DSCB so generated is DS1, DS2, ..., DS9, corresponding to the order of specification of the data set. The name DSx is assigned to the first word of the DSCB, the event conrol block. Fields within these DSCBs may be referenced symbolically with the expression:

```
  DSx+name
```

where name is a label defined in the DSCB equate table, DSCBEQU.

When overlay programs have been specified in the PROGRAM statement of an application program, a DSCB is created in the program header for each such overlay. Each of these can be referred to by the name PGMx where x is a number from 1 to 9 corresponding to the order of specification of the program name. Fields within these DSCBs may be referenced as PGMx+name where name is a label defined in the DSCB equate table, DSCBEQU.

DSCBs are automatically generated for data sets referenced by the DS and PGMS operands of the PROGRAM statement.

It is also possible to generate and use additional DSCBs within your program by coding DSCB statements. These DSCBs are named with the DS# operand.

<u>Syntax</u>

```
label       DSCB    DS#=,DSNAME=,VOLSER=,DSLEN=

Required:   DS#=,DSNAME=
Defaults:   VOLSER=null, DSLEN=0
Indexable:  none
```

Operands	Description
DS#=	The alphameric name which is used to refer to a DSCB in disk or tape I/O instructions. This name will be assigned to the first word (ECB) of the generated DSCB. Specify 1 to 8 characters.
DSNAME=	The data set name field within the DSCB. Specify 1 to 8 characters.
VOLSER=	The volume label to be assigned to the volume label field of the DSCB. Specify 1 to 6 characters. A null entry (blanks) will be generated if VOLSER is not specified. Note, however, that if the DSCB is for a tape data set, VOLSER must be specified prior to DSOPEN. Also for tape data sets, if there is no volume label, then the 1 - 6 digit tape drive ID must be supplied. The tape drive ID is assigned with the TAPE configuration statement during system generation.
DSLEN=	The size of the referenced direct access space. If no number is specified, this value will be set to 0. This parameter is not used if the DSOPEN routine will be used to open the DSCB.

When a data set is defined using the DSCB statement it must be opened before attempting disk or tape I/O operations such as READ or WRITE. The routines DSOPEN and $DISKUT3 are provided for this purpose. DSOPEN must be copied into your program with the COPY instruction and then invoked with the CALL instruction. The $DISKUT3 is invoked with the LOAD instruction. For more information on DSOPEN refer to the System Guide "Advanced Topics" section.

Example

```
         DSCB      DS#=INDATA,DSNAME=MASTER,
                   VOLSER=EDX003
```

ECB

Task Control

The ECB statement generates a 3-word event control block (ECB).

Normally this statement is not needed to write application programs if the program is to be assembled by the host or Series/1 macro assemblers. In this case Event Control Blocks are automatically generated for you as a consequence of your naming an event in a POST instruction. However, it may be used for special purposes such as controlling their location within a program. You must explicitly code necessary ECBs in programs to be assembled by $EDXASM, except for those created by specifying EVENT in a PROGRAM or TASK statement. Also, when coding an ECB instruction the label of the ECB must be the same as the event name.

A maximum of 25 ECB statements may be coded in a program. If more than 25 ECBs are required, they must be coded using the DATA statement. (See the example following the syntax description.)

Syntax

```
label       ECB   code

Required:   label
Defaults:   code = -1
Indexable:  none
```

Operands	Description
code	Initial value of the code field (word 1). If this word is non-zero when a WAIT is issued, no wait occurs unless the WAIT has RESET coded.

Example

```
ECB1         ECB
```

is equivalent to coding,

```
ECB1         DATA      F'-1'
             DATA      2F'0'
```

Note that ECB is not an executable statement and should not be placed between executable instructions.

END

Task Control

The END statement must be the last statement coded in your program.Unpredictable results may occur when the END statement is not coded.

<u>Syntax</u>

```
blank        END

Required:    none
Defaults:    none
Indexable:   none
```

Operands	Description
none	none

ENDATTN

Task Control

The ENDATTN statement ends an attention interrupt handling routine, as described under ATTNLIST, and is the last statement of that routine.

An attention interrupt handler should be a short routine used to provide an operator with terminal keyboard initiation or control of application routines.

Syntax

```
label       ENDATTN

Required:   none
Defaults:   none
Indexable:  none
```

Operands	Description
none	none

Example: See ATTNLIST instruction and also "Example 7: A Two Task Program With ATTNLIST" on page 395.

ENDPROG

Task Control

The ENDPROG statement must be the next to the last statement in a user program. The last statement must be END.

Syntax

```
blank        ENDPROG

Required:    none
Defaults:    none
Indexable:   none
```

Operands	Description
none	none

ENDTASK

Task Control

The ENDTASK statement defines the end of a block of instructions associated with a task. Each task, except the initial task, requires one ENDTASK as its final statement. When this instruction is executed, the task will be detached. If another ATTACH is issued, execution will resume at the initial instruction of the task.

ENDTASK actually generates two instructions: DETACH and GOTO start where start is the label of the first instruction to be executed when the task is first attached.

Syntax

```
label       ENDTASK     code,P1=

Required:   none
Defaults:   code=-1
Indexable:  none
```

Operands	Description
code	The posting code to be inserted in the terminating ECB ($TCBEEC) of the task being detached. Additional information on this statement can be found in the Internal Design manual (LY34-0168).
P1=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

ENQT

Terminal I/O

The ENQT instruction acquires exclusive access to a terminal until a DEQT is executed. ENQT is also used to establish terminal configuration parameters, such as the limits and mode of a logical screen, which will be in effect during the period of exclusive access.

Note: As part of the LOAD function, a DEQT of the terminal currently in use by the loading program is performed. You should allow for this circumstance in coding the program which issues the LOAD instruction.

Syntax

```
label       ENQT  name,BUSY=,P1=

Required:   none
Defaults:   name=terminal from which the issuing program
                 was loaded
Indexable:  none
```

Operands	Description
name	In general, this parameter is the label of an IOCB statement defining the terminal to be accessed, and this form would be used to establish temporary terminal configuration parameters. However, two special names are recognized: $SYSLOG and $SYSPRTR. When one of these names is used, the terminal acquired is the one whose TERMINAL statement has that label. If this parameter is not specified, or if no terminal with the indicated name exists, then access defaults to the terminal from which the program was loaded.
	If this operand is an IOCB with asscociated logical screen limits, internal terminal I/O counters are reset to start output at the top of the working area. The output starts at the top of the working area although the cursor is not immediately moved to this position. Before doing any input, the cursor can be moved by issuing TERMCTRL DISPLAY.

BUSY=	The terminal to which the ENQT instruction is directed may have been acquired by another task or may be in use by a supervisor utility function. The requesting task is then placed in a queue, waiting for the device, and its operation is suspended until all other users preceding it have been serviced. The BUSY operand allows the program to detect such a busy condition before it is placed in the queue. Code BUSY with the label of the instruction where execution is to proceed to if the terminal is in use.
P1=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

ENTRY

Program Module Sectioning

The ENTRY statement defines one or more labels as being entry points within a program module. These entry point labels may be referenced by instructions in other program modules that are link-edited with the module which defines the entry label. The program modules which reference the label must contain either a EXTRN or WXTRN statement for the label.

Syntax

```
blank       ENTRY  one or more relocatable symbols
                   separated by commas

Required:   one symbol
Defaults:   none
Indexable:  none
```

Operands	Description
	One or more symbols that appear as statement labels within the program module.

EOR

Data Manipulation

The EOR instruction (exclusive OR) makes a logical comparison of two bit-strings and provides a result, bit by bit, of 1 or 0. If the inputs are the same, the result is 0. If the inputs are not alike, the result is 1. If the entire input fields are identical, the entire resulting field will be 0. If one or more bits differ, the resulting field will contain a mixture of 0s and 1s.

Syntax

```
label       EOR     opnd1,opnd2,count,RESULT=,
                    P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   count=(1,WORD),RESULT=,opnd1
Indexable:  opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies; it cannot be a constant.
opnd2	The value to be compared to the first operand. Either the name of a variable or an explicit constant may be specified. Opnd2 can only be a BYTE, WORD, or DWORD.
count	The number of consecutive variables upon which the operation is to be performed. The maximum value allowed is 32767.

The count operand can include the precision of the data. Because these operations are parallel (the two operands and the result are implicitly of like precision), only one precision specification is required. That specification may take one of the following forms:

BYTE -- Byte precision
WORD -- Word precision
DWORD -- Doubleword precision

TYPE=	The type of data to be erased. TYPE=DATA: Only unprotected characters are erased. TYPE=ALL: Both protected and unprotected characters are erased.
SKIP=	The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), it is divided by the page size, and the remainder is the number of lines skipped.
LINE=	This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. For static screens, the I/O operation will take place on the line specified. In any case, if the value exceeds the last usable line number, it is divided by the logical page size, and the remainder is used as the line number.
SPACES=	The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line.

Whenever LINE or SKIP is specified on an instruction, the current indent is reset to zero (carriage return). For static screens in particular, specification of both LINE and SPACES designates a character position in two-coordinate form. If SPACES is specified without LINE or SKIP, then the indent value is increased by the value specified.

Example

```
ERASE   4,MODE=FIELD,TYPE=DATA
ERASE   LINE=0,SPACES=0,MODE=SCREEN,TYPE=ALL
ERASE   LINE=1,MODE=LINE,TYPE=ALL
```

EXIO

EXIO Control

EXIO is used to request execution of a command in a user-defined IDCB.

Syntax

```
label       EXIO     idcbaddr,ERROR=,P1=

Required:   idcbaddr
Defaults:   none
Indexable:  idcbaddr
```

Operands	Description
idcbaddr	The address of an IDCB.
ERROR=	The label of the first instruction executed if an error occurs during execution of this command. This instruction will not be executed if an error is detected at the occurrence of an interrupt caused by the command. The condition code (ccode) returned at interrupt time is posted in an ECB (see the EXOPEN instruction). A 'Device Busy' bit is set on by the EXIO instruction if a START command is executed. It is reset after the device interrupts if the operation is complete. If a device fails to interrupt or complete an operation, it will be necessary to reset the 'Device Busy' bit so that another command may be executed. The device busy bit can be reset by issuing an EXIO instruction to the appropriate IDCB followed by an IDCB instruction with COMMAND=RESET.
P1=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Note: For a list of instruction and interrupt condition codes, see the EXOPEN instruction and Figure 7 on page 131 and Figure 8 on page 132.

require a position for the sign. If the number has less than w digits, the leftmost print positions are filled with blanks. If the quantity is negative, the position preceding the leftmost digit contains a minus sign.

The following examples show how each of the quantities on the left is converted, according to the specification 'I3':

Internal Value	Value in the Buffer
721	721
-721	***
-12	-12
8114	***
0	0
-5	-5
9	9

Note that all error fields are stored and printed as asterisks.

Floating Point Numeric Conversion

General form: Fw.d

For F-type conversion, w is the total field length and d is the number of places to the right of the decimal point. For output, the total field length must include positions for a sign, if any, and a decimal point. The sign, if negative, is also loaded. For output, w should be at least equal to d + 2.

If insufficient positions are reserved by d, the number is rounded up. If excessive positions are reserved by d, zeros are filled in from the right for the insignificant digits.

If the integer portion of the number has less than w-d-1 digits, the leftmost print positions are filled with blanks. If the number is negative, the position preceding the leftmost digit contains a minus sign.

The following examples show how quantities are converted according to the specification F5.2:

Internal Value	Value in the Buffer
12.17	12.17
-41.16	*****
-.2	-0.20
7.3542	b7.35
-1.	-1.00
9.03	b9.03
187.64	*****

Notes:

1. 'b' represents a blank character stored in the text buffer.

2. Internal values are shown as their equivalent decimal value, although actually stored in floating-point binary notation requiring 2 or 4 words of storage.

3. All error fields are stored and printed as asterisks.

4. Numbers for F-conversion input need not have their decimal points appearing in the input fields (in the text buffer). If no decimal point appears, space need not be allocated for it. The decimal point is supplied when the number is converted to an internal equivalent; the position of the decimal point is determined by the format specification. However, if the position of the decimal point within the field is different from the position in the format specification, the position in the field overrides the format specification. For example, for a specification of F5.2, the following conversions would be performed:

Text Buffer Characters	Converted Internal Value
12.17	12.17
b1217	12.17
121.7	121.7

Floating Point Number Conversion (E-notation)

General form: Ew.d

For E-type conversion, w is the total field length and d is the number of places to the right of the decimal point. For output, the total field length must include sufficient positions for a sign, a decimal point, and space for the E notation (4 digits). For output, w should be at least equal to d + 6. For input, d is used for the default decimal position if no decimal is found in

```
((variable,count,type),-----)
or
(variable,-----)
or
((variable,count),-----)
or
((variable,type),-----)
```

where:

variable:	is the name of a variable or group of variables to be included.
count:	is the number of variables that are to be converted.
type:	is the type of variable to be converted.

S - Single-precision integer (default)
D - Double-precision integer
F - Single-precision floating-point
L - Extended-precision floating-point

The type will default to S for integer format data and to F for floating-point format data

format list

If you wish to refer to this format statement from another GETEDIT instruction, then both the format and format list operands must be coded. Refer to the FORMAT statement for coding instructions. This operand is not allowed if the program is compiled with $EDXASM.

ERROR= The name of a user's routine to branch to if an error is detected during the GETEDIT execution. Errors that might occur causing this action to take place are:

- Use of an incorrect format list.
- Field omitted (attempt is made to convert the rest)
- Not enough data in input text buffer to satisfy the Data List.

- Conversion error (value too large).
- Only the highest value return code is returned to you.

  The error indicators (return codes) are listed in the description of the CONVTD instruction.

  An error indicator is returned to you, whether or not you coded the ERROR parameter.

ACTION= IO causes a READTEXT instruction to be executed prior to conversion.

STG causes the conversion of a text buffer that has been previously obtained. The data must be in EBCDIC.

SCAN= FIXED - Data elements in the input text buffer must be in the format described in the format statement. That is, if a field width is specified as 6, then there are 6 EBCDIC characters used for the conversion. Leading and trailing blanks are ignored.

FREE - Data elements in the input text buffer must be separated by delimiters: blank, comma, or slash. If A format type items are included, they must be enclosed in apostrophes, for example, 'xyz'. This allows the inclusion of any alphameric characters except the apostrophe.

SKIP= The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page size and the remainder is the number of lines skipped.

LINE= This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used as the line number.

GETVALUE

Terminal I/O

GETVALUE is used to read one or more integer numeric values, or a single floating-point value, entered by the terminal operator. The values may be decimal or hexadecimal, of single or double-precision or floating-point. If an invalid character is entered, it acts as a delimiter. The printing of an associated prompt may be unconditional, or it may be conditional upon the absence of advance input.

Syntax

```
label       GETVALUE   loc,pmsg,count,MODE=,PROMPT=,
                       FORMAT=,TYPE=,SKIP=,LINE=,
                       SPACES=,P1=,P2=,P3=

Required:   loc
Defaults:   MODE=DEC,PROMPT=UNCOND,count=1 (word)
            FORMAT=(6,0,I),TYPE=S
            SKIP=0,LINE=current line,SPACES=0
Indexable:  loc,pmsg,SKIP,LINE,SPACES
```

Operands	Description
loc	Name of the variable to receive the input value. If the number of values requested is greater than one, then successive values are stored in successive words or doublewords.
pmsg	Name of a TEXT statement or an explicit text message enclosed in apostrophes. This defines the prompting message which will be issued according to the value of the PROMPT keyword.
count	Specify the number of integer values to be entered. The precision specification may be substituted for the count specification, in which case the count defaults to 1, or it may accompany the count in the form of a sublist: (count,precision). Precision may be either WORD (the default) or DWORD (doubleword).

With conditional prompting in effect, the absence of advance input causes the prompt message to be issued. Once a prompt message has been issued, however, zero or more values may be entered. Omitted values leave the corresponding internal variables unchanged. Permitted delimiters between values are the characters slash, comma, period, or blank. At completion of the instruction, the number of values entered is stored at taskname+2.

MODE=	Use MODE=HEX for hexadecimal input. The default (MODE=DEC) is decimal.
PROMPT=	Code PROMPT=COND or PROMPT=UNCOND (PROMPT=UNCOND is the default)

FORMAT= This parameter is used to specify external formatting for the input of a single value. The count parameter is ignored. The format is specified as a 3-element list (w,d,f), defined as follows:

w	A decimal value equal to the maximum field width in bytes expected from the terminal.
d	A decimal value equal to the number of bytes to the right of an assumed decimal point. (An actual decimal point in the input will override this specification.) For integer variables, code this value as zero.
f	Format of the input data

I	integer
F	floating-point F format
E	floating-point E format

TYPE= Use this operand only in conjunction with FORMAT=.

S	Single-precision integer (1 word)
D	Double-precision integer (2 words)
F	Single-precision floating-point (2 words)
L	Extended-precision floating-point (4 words)

SKIP= The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page size, and the remainder is the number of lines skipped.

LINE=	This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used as the line number.
SPACES=	These parameters may be used to specify the location within the logical page at which input is to begin, if that location differs from the current line and indent. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
          GETVALUE  DATA,MESSAGE
          GETVALUE  DATA2,'@ENTER A: ',PROMPT=COND
          GETVALUE  DATA3,MSG,5,MODE=HEX
              .
              .

MESSAGE   TEXT   'ENTER YOUR AGE'
MSG       TEXT   'DATA: '
DATA      DATA    F'0'
DATA2     DATA    F'0'
DATA3     DATA    5F'0'
```

GIN

GIN provides interactive graphical input. It rings the bell, displays cross-hairs, waits for the operator to position the cross-hairs and key in any single character, returns the coordinates of the cross-hair cursor, and optionally returns the character entered by the user. Cursor coordinates are unscaled. The PLOTGIN instruction obtains coordinates scaled by the use of a PLOTCB control block. (See "PLOTGIN" on page 210 for the format of a PLOTCB).

Syntax

```
label       GIN     x,y,char,P1=,P2=,P3=

Required:   x,y
Defaults:   no character returned
Indexable:  none
```

Operands	Description
x,y	Locations for storage of coordinates of the cursor.
char	Location where character is to be stored. The character is stored in the right-hand byte; the left byte will be set to zero. If omitted, the character is not stored.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

IF

Program Sequencing

IF determines whether a relational statement or statement string is true or false, and then branches to a user specified address or passes control to true code or false code within the IF-ELSE structure.

Note: Because IF, ELSE, and ENDIF are usually coded together, the ELSE and ENDIF instructions are repeated here for your convenience.

| The use of parameters is not allowed on the IF statement.

Syntax

```
label       IF     statement

label       IF     statement,GOTO,loc

Required:  one relational statement
Defaults:  none
Indexable: data1 and data2 in each statement
```

Operands	Description
statement	A relational statement or statement string indicating the relationship(s) to be tested. Each statement is enclosed in parentheses. If GOTO is coded and the statement is true, the next instruction executed is defined by loc. If GOTO is not coded, THEN is assumed and the next instruction is determined by the IF-ELSE-ENDIF structure. If the condition is true, execution proceeds sequentially. The various forms of relational statements are fully described following "Program Sequencing Instructions" on page 34 and a number of examples are shown below.
GOTO	If the statement is true and GOTO is coded, control is passed to the instruction at loc. If the statement is false, execution proceeds sequentially.

loc	Used with GOTO to specify the address of the instruction to be executed if the statement is true. The instruction must be on a full-word boundary.
	Note: THEN can be coded after statement. This may be desired to comment the instruction for program readability.

ELSE

ELSE defines the start of the false code associated with the preceding IF instruction. The end of the false code is the next ENDIF instruction.

Syntax

```
label       ELSE

Required:   none
Defaults:   none
Indexable:  none
```

ENDIF

ENDIF indicates the end of an IF-ELSE structure. If ELSE is coded, ENDIF indicates the end of the false code associated with the preceding IF instruction. If ELSE was not coded, ENDIF indicates the end of the true code associated with the preceding IF instruction.

Syntax

```
label       ENDIF

Required:   none
Defaults:   none
Indexable:  none
```

Examples of IF, ELSE, and ENDIF

1. IF with GOTO

```
      IF    (A,EQ,B),GOTO,ANEB
```

2. Single IF

```
      IF     (C,NE,D)    or      IF      (C,NE,D),THEN
        :
        : (execute if C NE D)
        :
      ENDIF
```

3. IF with ELSE

```
      IF     (#1,EQ,1)
        :
        : (execute if #1 EQ 1)
      ELSE
        :
        : (execute if #1 NE 1)
      ENDIF
```

4. Double IF with ELSE

```
      IF     (A,EQ,B),AND,(C,EQ,D)
        :
        : (execute if A EQ B and C EQ D)
      ELSE
        :
        : (execute if either A NE B or C NE D)
      ENDIF
```

5. IF with nesting

```
IF   (A,EQ,B)      If A equals B and X is
 x1                greater than Y, instructions
  IF   (X,GT,Y)    x1, x2, and x3 will be executed.
    x2             If A equals B, but X is not
  ENDIF            greater than Y, instructions x1
 x3                and x3 are executed. If A does
ELSE               not equal B, only instruction x4
 x4                is executed.
ENDIF
```

Examples of relational statements

Relational statement	Comments
(A,EQ,0)	A equal to 0, WORD
(A,EQ,X'0022')	A equal to hex 22, WORD
(A,NE,B)	A not equal to B, WORD
(DATA1,LT,DATA2,WORD)	DATA1 less than DATA2, WORD
(CHAR,EQ,C'A',BYTE)	CHAR equal to 'A', BYTE
(XVAL,GT,Y,DWORD)	XVAL greater than Y, DWORD
((A,#1),EQ,1)	(A,#1) equal to 1, WORD
((A1,#1),LE,(B1,#2))	(A1,#1) LE (B1,#2), WORD
(#1,EQ,1)	#1 equal to 1, WORD
(#1,GT,#2)	#1 greater than #2, WORD
((C,#2),EQ,CHAR,BYTE)	(C,#2) equal to CHAR, BYTE
(A,EQ,B,8)	A equal to B, 8 bytes
((BUF,#1),NE,DATA,3)	(BUF,#1) not equal to DATA, 3 bytes
(F1,GT,0,FLOAT)	F1 greater than 0, FLOAT
(L2,LT,L3,DFLOAT)	L2 less than L3, EXTENDED FLOAT
((BUF,#1),LE,1,FLOAT)	(BUF,#1) less than or equal 1, FLOAT

Examples of relational statement strings

```
(A,EQ,B),AND,(A,EQ,C)
(A,NE,1),OR,(D,EQ,E,DWORD),AND,(#1,NE,14)
(F,EQ,G,8),AND,(#1,EQ,#2),AND,(X,EQ,1),OR,(RESULT,GT,0)
(DATA,EQ,C'/',BYTE),OR,(DATA,EQ,C'*',BYTE)
((BUF,#1),NE,(BUF,#2)),OR,(#1,EQ,#2)
```

IOCB

Terminal I/O

IOCB defines a terminal name and configuration parameters for use with the ENQT instruction. Additional information on the configuration parameters can be found under the TERMINAL system configuration statement in the System Guide

Syntax

```
label       IOCB  name,PAGSIZE=,TOPM=,BOTM=,LEFTM=,
                  RIGHTM=,SCREEN=,NHIST=,OVFLINE=,BUFFER=

Required:   none
Defaults:   see discussion below
Indexable:  none
```

Operands	Description
name	The name of a terminal as defined by the label on a TERMINAL statement. See the System Configuration section of the System Guide for a description of the TERMINAL statement. This operand generates an 8-character EBCDIC string, padded as necessary with blanks, whose label is the label on the IOCB statement. It may therefore be modified by the program. If unspecified, the string is blank and implicitly refers to the terminal from which the program was loaded.
PAGSIZE=	This operand is as defined for the TERMINAL statement. Its default is the value assigned in that statement. If this parameter is modified, BOTM= must be between TOPM= plus NHIST=, and PAGSIZE-1. Otherwise, unpredictable results will occur.
TOPM=	As defined for TERMINAL. The default is 0.
BOTM=	As defined for TERMINAL. The default is PAGSIZE-1.
LEFTM=	As defined for TERMINAL. The default is 0.

RIGHTM=	As defined for TERMINAL. If the BUFFER statement is not specified, the default is LINSIZE-1. If the BUFFER statement is specified, the value you specify should be one less than the buffer size value.
SCREEN=	Either SCREEN=ROLL or SCREEN=STATIC, as defined for TERMINAL. The default is ROLL.
NHIST=	As defined for TERMINAL. The default is 0.
OVFLINE=	As defined for TERMINAL. The default is NO.
BUFFER=	If the application requires a temporary I/O buffer different than that defined by the LINSIZE parameter on the TERMINAL statement, then set this operand with the label of a BUFFER statement allocating the desired number by bytes. The buffer size temporarily replaces the LINSIZE value and is also the maximum amount that can be read or written at a time. For data entry applications which require full screen data transfers, for example, this obviates the need for allocation of a large buffer within the resident supervisor.

Note that when the buffer size is greater than the 80-byte line size of the 4978/4979 display, all data transfers take place as if successive lines of the display were concatenated. Screen positions are still designated, however, by the LINE and SPACES parameter with respect to an 80-byte line.

If the buffer size is less than the 80-byte linesize of the 4978/4979 display, the logical screen boundaries are adjusted accordingly. The logical screen refers to the part of the physical screen from which you can read or write.

If RIGHTM is not specified or has a value greater than the buffer size, it is adjusted to one less than the buffer size value. Portions of the screen outside this range are not accessible by the application program.

If the temporary buffer is not directly addressed by a terminal I/O instruction, then it acts as a normal system buffer of size RIGHTM+1; it may also be used, however, for direct terminal I/O. Direct terminal I/O occurs when the buffer defined by an active IOCB is directly addressed by a PRINTEXT or READTEXT instruction; the data is transferred immediately and the new line character is not recognized. When performing direct output operations the user must insert the output character count in the index word of the BUFFER prior to the PRINTEXT (output) instruction. This mode of operation allows the transfer of large blocks (larger than can be accommodated by a TEXT buffer) of data to and from buffered devices such as the 4978/4979 Display or buffered teletypewriter terminals. Upon execution of DEQT, the buffer defined by the TERMINAL statement is restored.

IODEF

Sensor Based I/O

IODEF is used to provide addressability for the Sensor Based I/O facilities which are referenced symbolically in an application program. The specific form used varies with the type of I/O being specified as shown below.

All IODEF statements of the same form (AI, AO, DI, DO, or PI) must be grouped together in the program and must be placed ahead of the SBIO instructions that reference them.

Each IODEF statement creates an SBIOCB control block. The contents of the SBIOCB is described in the Internal Design.

The IODEF statement generates a location into/from which data is read/written. You must create a separate IODEF for each task; different tasks cannot use the same IODEF statement.

The remainder of this description is divided into five parts to show the syntax for PI,DO,DI,AO, and AI. Because the operand definitions are common they are shown only once following the AI syntax.

Syntax

Process Interrupt

```
label    IODEF  PIx,ADDRESS=,BIT=,SPECPI=
                 or ADDRESS=,TYPE=BIT,BIT=,SPECPI=
                 or ADDRESS=,TYPE=GROUP,SPECPI=
```

Digital Output

```
label    IODEF  DOx,TYPE=GROUP,ADDRESS=
                 or TYPE=SUBGROUP,ADDRESS=,BITS=(u,v)
Syntax
```

Digital Input

```
label    IODEF  DIx,TYPE=GROUP,ADDRESS=
                 or TYPE=SUBGROUP,ADDRESS=,BITS=(u,v)
                 or TYPE=EXTSYNC,ADDRESS=
```

Analog Output

```
label       IODEF  AOx,ADDRESS=,POINT=

Defaults:   POINT=0
```

LASTQ

Queue Processing

LASTQ acquires entries from a queue, defined by DEFINEQ, on a last-in-first-out (LIFO) basis. Each time LASTQ is used, the last (most recent) entry is removed from the specified queue and returned to the user. The queue entry (QE) will then be added to the free chain of the queue.

<u>Syntax</u>

```
label        LASTQ  qname,loc,EMPTY=,P1=,P2=

Required:    qname,loc
Default:     none
Indexable:   qname,loc
```

<u>Operands</u>	<u>Description</u>
qname	The name of the queue from which the entry is to be fetched. The queue name is the label on the DEFINEQ instruction used to create the queue.
loc	The address of one word of storage where the entry is placed. #1 or #2 can be used.
EMPTY=	Use this operand to specify the first instruction of the routine to be invoked if "queue empty" condition is detected during the execution of this instruction. If this operand is not specified, control will be returned to the next instruction after the LASTQ and the user may test the task code word for a -1 indicating successful completion of the operation or a +1 if the queue is empty.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

<u>Example</u>: See the example following the NEXTQ instructions.

LOAD

Task Control

Note: Indexed Access Method LOAD is located under "LOAD" on page 344.

The LOAD instruction is used in one program to initiate the loading of another main program or program overlay from the program library on disk or diskette. The loaded program will run in parallel with, and independently of, the loading program, regardless of whether it is a main program or an overlay.

Data parameters and data set names may be passed to the loaded program. Also, the loading program may synchronize its own execution with that of the loaded program. Overlays must not contain TASK statements.

A program may be loaded in two ways:

- As an independent program in its own contiguous storage area
- As an overlay program within the storage area allocated for the loading program

The advantages of the independent LOAD operation are:

- Main storage is allocated only when required
- More than one program may be loaded for simultaneous execution

The advantages of the overlay LOAD operation are:

- The availability of main storage can be guaranteed by the loading program since it is within its own storage area
- The loaded program will be brought into storage more quickly than by an independent LOAD

The task code word of the loading task may be tested to determine the result of the program load operation. The code word is refered to by the task name. The label of the task code word of the initial task is the name of the program. If this word is -1 the operation was successful. For the definition of error codes returned during the load process, see "Return Codes" later in this description.

EVENT= This is the symbolic name of an event (ECB statement) which is to be posted complete when the loaded program issues a PROGSTOP.

By issuing a LOAD and a subsequent WAIT for this event, the loading program may synchronize its own execution with that of the loaded program. The ECB must not be reset by either a WAIT (with RESET) or a RESET instruction.

Figure 10 on page 200 shows the flow of control in the two ways of loading a program.

Note: If this operand is specified, the loading program must ultimately WAIT for completion of the loaded program. If this is not done, a POST will be issued when the loaded program terminates even though the loading program may no longer be active, and unpredictable results can occur.

PART= This optional operand is used to specify cross partition loading of a program in a system containing more than 64K of storage. If PART is not coded, the program will be loaded in the same partition as the loading program.

Code PART='n' to specify the partition number into which to load the new program (n = 1 to 8).

Code PART=ANY to load the new program in any available partition.

Code PART='label' to point to a word in storage which contains the partition number in which to load the new program. Zero in the word pointed to by label is the same as PART=ANY.

PGMx is not valid with PART.

ERROR= Use this operand to specify the label of the first instruction of the routine to be invoked if an error condition occurs during the load process. If not specified, control is returned to the instruction following the LOAD instruction and the user may test for errors by testing the return code stored at the taskname (see PROGRAM/TASK).

STORAGE= Use this operand to override the value specified in the STORAGE operand coded on the PROGRAM statement of the program to be loaded. Some application programs will have a minimum dynamic storage requirement; be sure you know what it is before using this override. A value of 0 means that the STORAGE value specified in the loaded programs header is not to be overridden. STORAGE=0 is the default.

If the total storage required for the program and the dynamic increment is not available the LOAD request will fail. See the PROGRAM statement STORAGE operand for additional information on dynamic storage.

P2=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

MOVE

Data Manipulation

Operand 2 is moved to operand 1. If operand 2 is "immediate data", it must be an integer between -32768 and +32767 which will be converted to floating point, if necessary.

Syntax

```
label       MOVE     opnd1,opnd2,count,FKEY=,TKEY=,
                     P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   count=(1,WORD)
Indexable:  opnd1,opnd2
```

Operands	Description
opnd1	The name of the variable to which the operation applies; it cannot be a constant.
opnd2	This operand determines the value by which the first operand is modified. Either the name of a variable or an explicit constant can be specified. Opnd2 is moved to opnd1. If opnd2 is immediate data, it must be an integer between -32768 and +32767 which will be converted to floating point, if necessary. If opnd2 is immediate data, and byte precision is specified, only the low order byte of the immediate data is moved.
count	Specify the number of consecutive variables upon which the operation is to be performed. A symbol cannot be used for count. The maximum value allowed for the count operand is 32767. Note: For all precisions other than BYTE, opnd1 and opnd2 must specify even addresses.

The count operand can include the precision of the data. Since these operations are parallel (the two operands and the result are implicitly of like precision) only one precision specification is required. That specification may take one of the following forms:

BYTE	-- Byte precision
WORD	-- Word precision
DWORD	-- Doubleword precision
FLOAT	-- Single-precision floating-point
DFLOAT	-- Extended-precision floating-point

The default precision is WORD.

The precision specification may be substituted for the count specification, in which case the count defaults to 1, or it may accompany the count in the form of a sublist: (count,precision). For example, MOVE A,B,BYTE and MOVE A,B,(1,BYTE) are equivalent.

FKEY= This operand provides a cross partition capability for opnd2 of MOVE. FKEY designates the address key of the partition containing opnd2 (The address key is one less than the partition number). FKEY can specify either an immediate value from 0 to 7 or the label of a word containing a value from 0 to 7. If FKEY is not specified, opnd2 is in the same partition as the MOVE instruction. If FKEY is specified, opnd2 cannot be immediate data or an index register. However, it can contain an index register if in the (parameter,#r) format.

TKEY= This operand provides a cross partition capability for opnd1 of MOVE. TKEY designates the address key of the partition containing opnd1 (the address key is one less than the partition number). TKEY can specify either an immediate value from 0 to 7 or the label of a word containing a value from 0 to 7. If TKEY is not specified, opnd1 must be in the same partition as the MOVE instruction. If TKEY is specified, opnd1 cannot be an index register. However, opnd1 can contain an index register if it is of the format (parameter,#r).

If TKEY is specified and opnd2 is immediate data, the immediate data is always 1 word in length regardless of any precision specification. However, a precision specification plus length is used in determining the total amount of data to be moved. Refer to Address Indexing Feature for further information.

Note: Refer to the _System Guide_ topic on "Cross-Partition Services" for additional information on the use of cross-partitions functions.

Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.
	Note: P3= will only alter the count operand. The precision specification will not be altered.

Example

```
MOVE    A,B                       move word, B to A

MOVE    TEXT,c' ',(64,BYTE)       move EBCDIC blank to
                                  64-byte field

MOVE    V1,V2,16                  move V2 to V1, 16 words

MOVE    SAVE,#1                   index register 1 to SAVE

MOVE    #2,INDEX                  set index register 2
                                  from INDEX

MOVE    D,C,(4,DWORD)             C to D, 4 doublewords

MOVE    F2,F1,(1,FLOAT)           F1 to F2, single-precision
                                  floating-point

MOVE    LR,L1,(6,DFLOAT)          L1 to LR, 6 extended float-
                                  ing point numbers (24 words)

MOVE    (BUF,#1),0,(10,FLOAT)     10 floating-point zero values
                                  to starting address (BUF,#1)

MOVE    HERE,$START,FKEY=0        move one word from $START in
                                  partition one to HERE

MOVE    (0,#1),#2,TKEY=KEY        move contents of #2 to a
                                  word in another partition
                                  at the address specified
                                  by #1

MOVE    ($NAME,#1),C'  ',         moves blanks into $NAME field
        (8,BYTES),TKEY=0          in partition 1 (opnd2 must be
                                  a word immediate value)
```

MOVEA

Data Manipulation

The address of operand 2 is moved to operand 1.

Syntax

```
label       MOVEA     opnd1,opnd2,P1=,P2=

Required:   opnd1,opnd2
Defaults:   none
Indexable:  opnd1
```

Operands	Description
opnd1	The name of the variable in which the address of opnd2 is stored.
opnd2	This operand determines the address value that is placed in opnd1.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
MOVEA  PTR,A        move address of A into PTR
MOVEA  PTR,B+4      move address of (B)+4 into PTR
```

MULTIPLY

Data Manipulation

Signed multiplication of operand 1 by operand 2. The instruction may be abbreviated MULT.

Note: An overflow condition in the MULIPLY instruction stores a hexadecimal value of 8000 in the task code word.

Syntax

```
label       MULTIPLY   opnd1,opnd2,count,RESULT=,PREC=,
                       P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   count=1,RESULT=opnd1,PREC=S
Indexable:  opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies; it cannot be a constant.
opnd2	This operand determines the value by which the first operand is modified. Either the name of a variable or an explicit constant may be specified.
count	Specify the number of consecutive variables upon which the operation is to be performed. The maximum value allowed is 32767.
RESULT=	This operand may optionally be coded with the name of a variable or vector in which the result is to be placed. In this case the variable specified by the first operand is not modified.
PREC=XYZ	Where X applies to opnd1, Y to opnd2, and Z to the result. The value may be either S (single-precision) or D (double-precision). 3-operand specification may be abbreviated according to the following rules:

- If no precision is specified, then all operands are single-precision.
- If a single letter (S or D) is specified, then it applies to the first operand and result, with the second operand defaulted to single-precision.
- If two letters are specified, then the first applies to the first operand and result, and the second to the second operand.

Px= Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Mixed-precision Operations: Allowable precision combinations for multiply operations are listed in the following table:

opnd1	opnd2	Result	Abbreviation	Remarks
S	S	S	S	default
S	S	D	SSD	-
D	S	D	D	-
D	D	D	DD	-

Example

```
  MULT   C,D,RESULT=E,PREC=SSD    double-precision product
  MULT   A,10,PREC=D              double precision variable A
                                  is multiplied by 10

  MULT   X,10,2                   the single-precision variables
                                  at X and X+2 are each
                                  multiplied by 10.
```

PRINTEXT

Terminal I/O

PRINTEXT is used to write a message to the terminal and to control forms movement. Forms control is always executed before the message is written.

Syntax

```
label       PRINTEXT  msg,SKIP=,LINE=,SPACES=,XLATE=,
                      MODE=,PROTECT=,P1=

Required:   At least one operand other than XLATE=,
            MODE= or PROTECT=
Defaults:   SKIP=0,LINE=(current line),SPACES=0,
            XLATE=YES,PROTECT=NO
Indexable:  msg,LINE,SKIP,SPACES
```

Operands	Description
msg	The name of a TEXT statement which defines the message to be printed or an explicit text message enclosed in apostrophes. If msg is the label of a BUFFER statement referenced by an active IOCB, then the output is direct, for example, the count is taken from the buffer index word at msg-4, the new line character is not recognized, and the operation is executed immediately. The direct I/O feature is useful for full control over a device, for example, to cause overstriking on a printer. The maximum line size of the terminal is established by the TERMINAL statement used to define the terminal when the system was configured. Refer to the TERMINAL statement in the System Guide for information on default sizes.
SKIP=	The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page

size, and the remainder is used in place of the specified value.

LINE= This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used in place of the specified value.

SPACES= The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line.

Whenever LINE or SKIP is specified on an instruction, the current indent is reset to zero (carriage return). For static screens in particular, specification of both LINE and SPACES designates a character position in 2-coordinate form. If SPACES is specified without LINE or SKIP, then the indent value is incremented by the value specified.

XLATE= To send character codes to the device as is, without translation by the system, code XLATE=NO. This option might be used, for example, to transmit graphic control characters and data. XLATE=YES causes translation of characters from EBCDIC to the terminals code.

Note: If the terminal requires that characters be sent in "mirror image", it is the user's responsibility to provide the proper bit representation if XLATE=NO is used.

MODE=

Code MODE=LINE if the text includes imbedded @ characters which are not to be interpreted as new line. For screens accessed in STATIC mode, MODE=LINE causes protected fields to be skipped over as the data is transferred to the screen. Protected positions do not contribute to the count. Do not code this parameter if @ characters are to be interpreted as new line.

PRINTIME

Terminal I/O

PRINTIME prints the time of day on the terminal. The value printed is in the form HH:MM:SS, according to a 24-hour clock, and is based upon the time value entered during the last $T entry of time.

Syntax

```
label        PRINTIME

Required:    none
Defaults:    none
Indexable:   none
```

Operands	Description
none	none

PRINTNUM

Terminal I/O

PRINTNUM is used to convert a floating point variable or one or more numeric integer variables to printable decimal or hexadecimal format and write them to the terminal with optional format control. Format specification can include, for integer data, the number of elements per line and the spacing between elements can be specified.

If the system buffer cannot accommodate the entire PRINTNUM value then any concatenated output is forced to the terminal before any other terminal I/O is executed.

Syntax

```
label       PRINTNUM  loc,count,nline,nspace,MODE=,FORMAT=
                      TYPE=,SKIP=,LINE=,SPACES=,PROTECT=
                      P1=,P2=,P3=,P4=

Required:   loc
Defaults:   count=1,nspace=1,MODE=DEC,PROTECT=NO,
            FORMAT=(6,0,I),TYPE=S,
            SKIP=0,LINE=current line,SPACES=0
            If nline is not specified, then it is
            determined by the terminal margin settings.
Indexable:  loc,SKIP,LINE,SPACES
```

Operands	Description
loc	Address of the first value to be printed. Successive values are taken from successive words or doublewords.
count	The number of values to be printed. The precision specification may be substituted for the count specification, in which case the count defaults to 1, or it may accompany the count in the form of a sublist: (count,precision). Precision may be either WORD (the default) or DWORD (double word).
nline	The number of values to be printed per line.
nspace	The number of spaces by which printed values will be separated.

MODE=	Code MODE=HEX for hexadecimal output. The default is decimal (MODE=DEC).

FORMAT= This operand is used to specify, in the form of a three-element list (w,d,f), the external format of a single variable to be printed. If this operand is specified integer or floating-point, then count, nline, nspace, and MODE are ignored. The format is defined as follows:

w	A decimal value equal to the field width in bytes of the data to be printed.
d	A decimal value equal to the number of significant digits to the right of the decimal point. For the integer format this value must be zero; for the floating-point F format it must be less than or equal to w-2, and for the floating-point E format less than or equal to w-6.
f	Format of the output data

I	Integer
F	Floating-point F format
E	Floating-point E format

TYPE= This operand is used to specify the type of the internal variable to be printed. Used only in conjunction with the FORMAT operand.

S	Single-precision integer (1 word)
D	Double-precision integer (2 words)
F	Single-precision floating-point (2 words)
L	Extended-precision floating-point (4 words)

SKIP= The number of lines to be skipped before the operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page size, and the remainder is used in place of the specified value.

LINE= This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size and the remainder is used in place of the value.

SPACES=	The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line. Whenever LINE or SKIP is specified on an instruction, the current indent is reset to zero (carriage return). For static screens in particular, specification of both LINE and SPACES designates a character position in 2-coordinate form. If SPACES is specified without LINE or SKIP, then the indent value is incremented by the value specified.
PROTECT=	Code PROTECT=YES to write protected characters to a device for which this feature is supported.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
  PRINTNUM  A

  PRINTNUM  BUF1,10

  PRINTNUM  AX,MODE=HEX

  PRINTNUM  BUF2,10,5,3

  PRINTNUM  BZ,(10,DWORD),MODE=HEX
```

PROGRAM

Task Control

PROGRAM is used to define basic parameters of a user program. PROGRAM is the first statement of every user program. When program assembly is to be done by $EDXASM, the PROGRAM statement may be omitted when assembling a subprogram. (See the MAIN operand for the definition of a subprogram.) When program assembly is to be done by the Host or Series/1 macro assemblers, the PROGRAM statement must be coded even for subprograms.

Syntax

```
taskname   PROGRAM start,priority,EVENT=,
           DS=(dsname1,...,dsname9),PARM=n,
           PGMS=(pgmname1,...,pgmname9),TERMERR=,
           FLOAT=,MAIN=,ERRXIT=,STORAGE=,WXTRN=

Required:  taskname,start (except when MAIN=NO)
Defaults:  priority=150,PARM=0,FLOAT=NO,MAIN=YES,
           STORAGE=0,WXTRN=YES
Indexable: none
```

Operands	Description
taskname	The name to be assigned to the primary task of the program. A system control block is generated for each task in an application program. This is known as the Task Control Block or TCB. The first word of the TCB is assigned the name specified in the taskname operand. This word is known as the 'task code word' and has a special significance in program operation. For example, in I/O operations it is used for storing a return code for the user. Thus, the task name may be used in an IF instruction to test for a successful completion of an I/O operation.
start	The label of the first instruction to be executed in your program. The instruction must be on a full word boundary.

priority	The priority of the program's primary task. Priorities separate tasks according to their relative critical real time needs for processor time. The range is from 1 (highest priority) to 510 (lowest priority). Priorities 1-255 imply foreground and are executed on hardware interrupt level 2. Priorities 256-510 imply background and are executed on interrupt level 3.
EVENT=	EVENT=name is used to name the event which will be posted when the initial task is detached. It must be defined only if another task will issue a WAIT for this event. This event name must not be defined explicitly by an ECB because it is generated automatically. An error message is printed at the end of the program if the event name is defined more than once.
DS=	Names of 1-9 disk, diskette, or tape data sets to be used by this program. Each name is composed of 1-8 alphameric characters, the first of which must be alphabetic.

One DSCB is generated in the program header for each data set specified in the DS parameter of the PROGRAM statement. The name of each DSCB so generated is DS1, DS2, ..., DS9, corresponding to the order of specification of the data set. The name DSx is assigned to the first word of the DSCB, the event control block. Fields within the DSCB may be referenced symbolically with the expression:

```
DSx+name
```

where name is a label defined in the DSCB equate table, DSCBEQU.

All tape data sets are of the form (DSN,VOLUME). The specification of tape data sets is dependent upon the type of label processing being done.

For standard label (SL) processing the DSN is the data set name as it is specified in the HDR1 label. VOLUME is the volume serial as it is specified in the VOL1 label.

When doing no label (NL) processing or bypass label processing (BLP) the volume must be specified as the 1 - 6 digits that represent the tape unit ID. The tape unit ID was assigned at system generation time. The DSN is ignored during NL or BLP processing but it must be supplied for syntax checking purposes. It also provides identification of the data set for things like error logging.

If more than one disk or diskette logical volume is being used, a volume label must be specified if the data set resides on other than the IPL volume. The data set name and volume are separated by a comma and enclosed in parentheses. In addition, the entire list of data set/volume names are enclosed in a second set of parentheses. For example:

```
   ...,DS=((ACTPAY,EDX001),(DSDATA2,EDX003))
```

references the data set ACTPAY on volume EDX001 and DSDATA2 on volume EDX003. If a volume is not specified, the default is the IPL volume.

When one data set is used and it is in the IPL volume, no parentheses are required. For example:

```
  DS=CUSTFIL
```

When more than one data set is used and they reside in the IPL volume, the data set names are separated by commas and enclosed in parentheses. For example:

```
  DS=(CUSTFIL,VENDFIL)
```

Four special data set names are recognized: ??, $$EDXLIB, and $$ or $$EDXVOL. A data set control block (DSCB) is created just as for any other data set name. However, special processsing occurs when the program is loaded for execution.

If the sequence '??' is used as a data set name, the final data set name and volume specification is done at program load time. If the program is loaded by another program, this information must be contained in the DS operand of the LOAD instruction. If the program is loaded using the system command '$L', the system will query the operator for these names. If the specified sequence is of the form

```
    DS = ((string,??))
```

where 'string' is 1-8 alphanumeric characters the user will be given a prompt message:

```
    string(NAME,VOLUME)
```

If the specified sequence is of the form

DS = ??

the user will then be given a prompt message

DSn(NAME,VOLUME):

where 'n' is a digit from 1 to 9.

If '$$EDXLIB is used as a data set name, the library directory of the specified volume is opened for processing. Note that record 1 contains a directory control entry and the first seven directory member entries. This is useful for the creation of utility programs or for "do it yourself" data set access. Update of the directory by user programs is not recommended since doing so incorrectly could cause the loss of some or all of the data sets in the volume.

If $$EDXVOL or $$ is used as a data set name, the entire volume is opened for processing as if it were a single data set. The library directory and any data sets on the volume are accessible. Note that record number 1 and 2, of a primary volume, can contain IPL text, and record number 4, of a diskette, contains the volume label. This is useful if the DISK statement defining the volume did not assign all available space to a library. $$EDXVOL or $$ can also be used if the application program does not use Event Driven Executive data set facilities at all. Additionally, it can be used to reserve a DSCB in the program header so that it can be filled in and opened (using DSOPEN) after execution begins.

PARM= In this operand, n is a word count specifying the length of a parameter list to be passed to this program at load time. Each word in the list may be referenced by the symbolic name $PARMx where x is the word position number in the list beginning with 1. The maximum length of this list is 368 words less 19 for each data set name specified in the DS operand and each program overlay name specified in the PGMS operand.

This parameter is valid for programs to be loaded by a LOAD instruction. The list address is specified as an operand of that instruction. The list would be filled in by the loading program and there are no list — A description of the variables or locations which contain the input data, having the form:

```
((variable,count,type),----)
or
(variable,----)
or
((variable,count),----)
or
((variable,type),----)
```

where:

variable - is the name of a variable or group of variables that are to be converted to EBCDIC.

count - is the number of variables that are to be converted.

type - is the type of the variable to be converted

S	Single-precision integer (Default)
D	Double-precision integer
F	Single-precision floating-point
L	Extended-precision floating-point

Type will default to S for integer format data and to F for floating-point format data.

format list — A FORMAT list. If you wish to refer to this format statement from another PUTEDIT instruction, then both the format and format list operands must be coded. Refer to the FORMAT statement for coding instructions. This operand is not allowed if the program is assembled with $EDXASM.

ERROR= — The name of a user's routine to branch to if an error is detected during the PUTEDIT execution. Errors that might occur that will cause this action to take place are:

- Use of incorrect format list
- Not enough space in text buffer to satisfy the data list
- Only the highest value return code is returned to you.

The error indicators (return codes) follow:

Return Codes

Code	Description
-1	Successful completion
3	Conversion error

ACTION= IO causes a PRINTEXT to be executed following the data conversion.

STG causes the conversion and movement of data into a text buffer. No I/O takes place.

SKIP= The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page size, and the remainder is used in place of the specified value.

LINE= This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used in place of the specified value.

SPACES= The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. Whenever LINE or SKIP is specified on an instruction, the current indent is reset to zero

(BOTM-TOPM-NHIST), then it is divided by the page size, and the remainder is used in place of the specified value.

LINE= This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used in place of the specified value.

SPACES= The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line.

Whenever LINE or SKIP is specified on an instruction, the current indent is reset to zero (carriage return). For static screens in particular, specification of both LINE and SPACES designates a character position in 2 coordinate form. If SPACES is specified without LINE or SKIP, then the indent value is incremented by the value specified.

P1= Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
         QUESTION  TEXT3,YES=POINT1
         QUESTION  'DO IT AGAIN?',NO=EXIT
         QUESTION  'RESTART?',YES=INITIAL,NO=ENDP
             .
             .
  TEXT3  TEXT      'GO TO POINT1?'
```

RDCURSOR

Terminal I/O

RDCURSOR is effective only for IBM 4978/4979 terminals accessed in STATIC mode. It is used to store the cursor position (line number and indent relative to the logical screen margins) in user-specified variables. For more information on STATIC screens refer to "Terminal I/O Instructions" on page 44.

<u>Syntax</u>

```
label       RDCURSOR  line,indent

Required:   line,indent
Defaults:   none
Indexable:  line,indent
```

<u>Operands</u>	<u>Description</u>
line	The name of the variable in which the cursor position, relative to the top margin of the logical screen accessed, is to be stored. If the cursor lies outside the line range of the logical screen, then -1 is stored.
indent	The name of the variable in which the cursor position, relative to the left margin of the logical screen, is to be stored. If the cursor position is not within the left and right margins of the logical screen, then -1 is stored.

<u>Example</u>

```
RDCURSOR  LN,SP
RDCURSOR  (Y,#1),(X,#1)
```

READ

Disk/Tape I/O

READ is used to retrieve one or more records from a direct access or tape data set into a user storage buffer. It is your responsibility to ensure that sufficient buffer space has been defined. Direct access data sets can be read either sequentially or randomly. These data sets are read in 256-byte record increments.

Tape data sets are read sequentially only. A tape READ retrieves a record from 18 to 32767 bytes long, as specified by the blksize parameter.

Syntax

```
label       READ  DSx,loc,count,relrecno|blksize,
                  END=,ERROR=,WAIT=,P2=,P3=,P4=

Required:   DSx,loc
Defaults:   count=1,relrecno=0 or blksize=256,WAIT=YES
Indexable:  loc,count,relrecno or blksize
```

Operands	Description
DSx	x specifies the relative data set number in a list of data sets defined by the user on the PROGRAM statement. It must be in the range of 1 to n, where n is the number of data sets defined in the list. A DSCB name defined by a DSCB statement can be substituted for DSx.
loc	The label of the area into which the data is read.
count	The number of contiguous records to be read. If this field is set to 0 by the program, no I/O operation will be performed. A count of the actual number of records transferred will be returned in the second word of the task control block if WAIT=YES is coded. Note, however, if the incorrect blocksize was specified, the actual blocksize will be stored in the second word of the TCB, not the number of records transferred. If an end-of-data condition

occurs (fewer records remaining in the data set than specified by the count field) the system will first read the remainder and then an end-of-data return code will be set.

relrecno	This operand specifies the number of the record, relative to the origin of the data set, to be read. Numbering begins with 1. This parameter may be a constant or the label of the value to be used. A specification of 0 or default to 0 indicates a sequential READ. Note however, if 0 is specified, the end-of-data will be the physical end-of-data, but if relrecno defaults to 0 end-of-data will be the logical end-of-data.
	This disk READ operand cannot be used in the same instruction with the tape READ blksize operand. If the relrecno operand is specified, the READ instruction is considered a direct I/O operation. End-of data is recognized only at the physical end of the data set, not at the logical end.
	Sequential READs and WRITEs start with relative record 1 or the record number specified by a POINT instruction. The supervisor keeps track of sequential READs and WRITEs and increments an internal next record pointer for each record read or written in sequential mode (relrecno is 0). Direct READs and WRITEs (relrecno is not 0) may be intermixed with sequential operations, but these do not alter the next sequential record pointer used by sequential operations.
blksize	This operand determines the number of bytes to be read from a tape data set. The range is from 18 to 32767. The value can either be a constant or the label of the value to be used. If this operand is not coded, or if 0 is coded, the default value of 256 bytes will be substituted.
	The first word of the TCB will contain the return code for the READ operation. If the specified blksize does not equal the actual blksize, the ERROR path will be taken and the second word of the TCB will contain the actual blksize. Note, however, that the blksize is only stored in the second word of the TCB if WAIT=YES is coded, or WAIT is not coded and allowed to default to YES. If you code WAIT=NO and the blsksize specification is incorrect, you can check the $DSCBR3 field in the DSCB for the actual number of records read or the actual blksize.

Disk/diskette Return Codes

READ/WRITE return codes are returned in two places:

- The Event Control Block (ECB) named DSn, where n is the number of the data set being referenced.
- The task code word referred to by taskname.

The possible return codes and their meaning are shown in Figure 18 on page 321.

If further information concerning an error is required, it may be obtained by printing all or part of the contents of the Disk Data Blocks (DDBs) located in the Supervisor. The starting address of the DDBs may be obtained from the linkage editor map of the supervisor. The contents of the DDBs are described in the Internal Design. Of particular value are the Cycle Steal Status Words and the Interrupt Status Word save areas, along with the contents of the word which contains the address of the next DDB in storage.

Code	Description
-1	Successful completion.
1	I/O error and no device status present (This code may be caused by the I/O area starting at an odd byte address).
2	I/O error trying to read device status.
3	I/O error retry count exhausted.
4	Error on issuing I/O instruction to read device status.
5	Unrecoverable I/O error.
6	Error on issuing I/O instruction for normal I/O.
7	A 'no record found' condition occurred, a seek for an alternate sector was performed, and another 'no record found' occurred i.e., no alternate is assigned.
9	Device was 'offline' when I/O was requested.
10	Record number out of range of data set--may be an end-of-file (data set) condition.
11	Device marked 'unusable' when I/O was requested.
12	DSCB was not open; DDB address = 0

Figure 12. READ/WRITE return codes

Tape Return Codes

Code	Description
-1	Successful completion
1	Exception but no status
2	Error reading STATUS
4	Error issuing STATUS READ
5	Unrecoverable I/O error
6	Error issuing I/O command
10	Tape mark (EOD)
20	Device in use or offline
21	Wrong length record
22	Not ready
23	File protect
24	EOT
25	Load point
26	Uncorrected I/O error
27	Attempt WRITE to unexpired data set
28	Invalid blksize
29	Data set not open
30	Incorrect device type
31	Incorrect request type on close request
32	Block count error during close
33	EOV1 label encountered during close
76	DSN not found

Example

```
ABC      PROGRAM  START1,DS=((MYDATA,234567))
START1   READ     DS1,BUFF,1,327,END=END1,ERROR=ERR,WAIT=YES
```

This statement reads a single 327-byte record from a standard labeled (SL) tape. If an end of data set tapemark is detected, control is transferred to the statement named END1. If an error occurred, control transfers to the statement named ERR.

```
ABCD     PROGRAM  START2,DS=((MYDATA,234567))
START2   READ     DS1,BUFF2,2,327,END=END1,ERROR=ERR,WAIT=YES
```

This statement performs the same as the previous example except that 2 records are read into your storage buffer (BUFF2). BUFF2 must be 654 bytes in length.

READTEXT

Terminal I/O

READTEXT is used to read an alphameric text string entered by the terminal operator. The printing of an associated prompting message may be either unconditional or conditional depending upon the absence of advance input.

Input keyed in before the program has requested it (advance input) normally resides in the system buffer. Therefore, when a READTEXT is issued, the advance input is read immediately from the buffer. However, if an explicit or implicit terminal output operation occurs, the advance input is lost because the system buffer is used for terminal output.

An example of implicit terminal output could be the SPACES operand coded on a READTEXT statement. This could be the same READTEXT statement that you intended the advance input for.

Syntax

```
label       READTEXT  loc,pmsg,PROMPT=,ECHO=,TYPE=,
                      MODE=,XLATE=,SKIP=,LINE=,SPACES=

Required:   loc
Defaults:   PROMPT=UNCOND,ECHO=YES,TYPE=DATA,MODE=WORD,
            XLATE=YES,SKIP=0,LINE=current line,SPACES=0
Indexable:  loc,pmsg,SKIP,LINE,SPACES
```

Operands	Description
loc	This operand is normally the label of a TEXT statement defining the storage area which is to receive the data; the storage area may be defined by DATA or DC statements as well, but the format produced by the TEXT statement must be adhered to. In order to satisfy the length specification, the input will be either truncated or padded to the right with blanks as necessary.

If the length specification is greater than the system buffer size, then the length will be limited to the buffer size. If a user buffer is specified on the IOCB statement and you have issued an ENQT to the corresponding terminal, then the user buffer size will apply to the input length.

This operand may also be the label of a BUFFER statement referenced by an active IOCB statement. In this case the input is "direct;" the maximum input count is taken from the word at loc-2, imbedded blanks are allowed, and the final input count is placed in the buffer index word at loc-4.

The maximum line size for the terminal is established by the TERMINAL statement used to define the terminal when the system was configured. Refer to the TERMINAL statement in the System Guide for information on the default sizes.

pmsg	The name of a TEXT statement or an explicit text message enclosed in apostrophes. This defines the prompting message which will be issued according to the value of the PROMPT operand.
PROMPT=	Code PROMPT=COND (conditional) or the default PROMPT=UNCOND (unconditional). If conditional prompting is specified and the terminal user enters advance input, the message defined by the pmsg operand is not displayed. Unconditional prompting causes the message to be displayed without regard to the presence of advance input. Note: If PROMPT=COND is coded without specification of a prompt message, then the system will not wait for user input if advance input is not presented; instead, the receiving TEXT buffer is filled with blanks and its input count is set to 0.
ECHO=	Code ECHO=NO if the input text is not to be printed on the terminal. This operand is effective only for devices which require the processor to 'echo' input data for printing. Note: The specification PROTECT=YES is equivalent.
MODE=	Code MODE=WORD if the input operation is to be terminated by the entry of a blank character (space). Lower case input is automatically changed into upper case. Code MODE=LINE if the string to be read can include imbedded blanks. Lower case characters are left in lower case. Any portion of the input which extends beyond the count indicated in the receiving TEXT statement will be ignored and will not be retained as advance input.

When READTEXT is directed to a static logical screen, the input operation is normally terminated by the count being decremented to zero (the input buffer size), by the beginning of a protected field, or by the end of the logical line. However, if MODE=LINE, the TYPE operand will determine whether protected fields are skipped and whether they contribute to the count, and the input oper-

	ation may continue beyond the logical screen boundary to the end of the physical screen. In this case, input continues from the end of each physical screen line to the beginning of the next line.
TYPE	This parameter is used to specify the type of data to be transferred from 4978/4979 terminals. The default is TYPE=DATA. Only data fields are transferred. Code TYPE=ALL to transfer both protected and data (non-protected) fields. TYPE=MODDATA is used to transfer only those data fields which have been modified by the terminal operator (4978 only). Code TYPE=MODALL to transfer, along with each modified data field, the protected fields which precede it.
XLATE=	Code XLATE=NO if the input line is not to be translated to EBCDIC. Note that the character delete and line delete codes lose their special functions under this option, and that MODE=LINE is implied. <u>Note</u>: If the terminal is of the type that transmits characters in "mirror image" format, the characters will be placed in storage in that format if XLATE=NO is used. XLATE=YES causes the supervisor to translate the terminal's binary code to EBCDIC, the standard Series/1 representation of data.
SKIP=	The number of lines to be skipped before the next operation. If a current concatenated line has not been written, then the first skip causes output of that line. If the value specified is greater than or equal to the logical page size (BOTM-TOPM-NHIST), then it is divided by the page size, and the remainder is used in place of the specified value.
LINE=	This operand is used to specify the line at which the next I/O operation will take place. Code a number between 0 and the number of the last usable line on the page (BOTM-TOPM-NHIST). For hardcopy devices or roll screens, if the value specified is less than or equal to the current line number, then the forms will move to the specified line on the next page, otherwise to that line on the current

page. In any case, if the value exceeds the last usable line number, then it is divided by the logical page size, and the remainder is used in place of the specified value.

SPACES= The I/O position for a terminal or logical screen is defined by the line number and the position, within that line, of the typing element or cursor. The SPACES parameter is used to specify an increment to the cursor position. It does not imply over-printing with blank characters on display screens. If the specified value positions the cursor beyond the logical screen limits, the cursor is moved the excess number of spaces onto the next line. Whenever LINE or SKIP is specified on an instruction, the current indent is rest to zero (carriage return). For static screens in particular, specification of both LINE and SPACES designates a character position in 2-coordinate form. If SPACES is specified without LINE or SKIP, then the indent value is increased by the value specified.

RETURN

Program Control

RETURN is used in a subroutine to provide linkage back to the calling program. A subroutine can contain more than one RETURN instruction.

Syntax

```
label        RETURN

Required:    none
Defaults:    none
Indexable:   none
```

Operands	Description
none	none

SBIO

Sensor Based I/O

SBIO provides communication using analog and digital I/O. Many options provide flexibility. Optional automatic indexing is provided using the previously defined BUFFER statement. A buffer address in the SBIO instruction can be automatically updated after each operation. A short form of the instruction, omitting loc (data location) is provided. When used, a data address within the SBIOCB is implied. Options available with digital input and output provide PULSE output and the manipulation of portions of a group with the BITS=(u,v) keyword parameter.

SBIO instructions are hardware address independent. The actual operation performed is determined by the definition of the sensor address in the referenced IODEF statement.

The IODEF statement generates a location into/from which data is read/written. You must create a separate IODEF for each task; different tasks cannot use the same IODEF statement.

An INPUT/OUTPUT CONTROL BLOCK (SBIOCB) is automatically inserted into the user's program for each referenced sensor I/O device. It supplies necessary information to the supervisor. These control blocks each contain two items, a data I/O area and an ECB. When an SBIO instruction is executed, the supervisor either stores (AI and DI operations) or fetches (AO and DO operations) data from a location in the IOCB with the label equivalent to the referenced I/O point (for example, AI1, DI2, DO33, AO1). These locations may be referenced in the application program in the same manner as any other variable. This allows the user to use the short form of the SBIO instruction (for example, SBIO DI1), and subsequently reference DI1 in other instructions. It may also be convenient to equate a more descriptive label to the symbolic names (for example SWITCH EQU DI15). However, the SBIO instruction must use the symbolic name as described above.

Each control block also contains an ECB to be used by those operations which require the supervisor to service an interrupt and 'post' an operation complete. These include analog input (AI), process interrupt (PI), and digital I/O with external sync (DI/DO). For process interrupt, the label on the ECB is the same as the symbolic I/O point (for example PIx). For analog and digital I/O the label is the same as the symbolic I/O point with the suffix 'END' (for example DIxEND).

For brevity, operands common to all versions of SBIO are described here and not in the individual instruction descriptions.

SUBROUT

Program Control

SUBROUT is used to define the entry point of a subroutine. Up to five parameters may be specified as arguments in the subroutine. The subroutine must have a RETURN instruction to provide linkage back to the calling task. Nested subroutines are allowed, and a maximum of 99 subroutines are permitted per Event Driven Executive program. If a subroutine is to be assembled as an object module which can be link-edited, an ENTRY statement must be coded for the subroutine entry point name.

A subroutine may be called from more than one task. When called, the subroutine will execute as part of the calling task. Since subroutines are not re-entrant, it may be desirable to enforce serial usage of the subroutine using ENQ/DEQ instructions.

The TASK statement must not be coded in a subroutine.

Syntax

```
label       SUBROUT name,par1,...,par5

Required:   name
Defaults:   none
Indexable:  none
```

Operands	Description
name	Name of the subroutine.
par1,...	Names used within the subroutine for arguments or parameters passed from the calling program. These names must be unique to the whole program. All parameters defined outside the subroutine are known within the subroutine. Thus, only parameters which may vary with each call to a subroutine need to be defined in the SUBROUT instruction. These parameters are defined automatically as single precision values.

For instance, assume two calls to the same subroutine. At the first, parameters A and C are to be passed, while at the second, B and C are to be passed. Because C is common to both, it need not be defined in the SUBROUT statement. However, a new parameter D would be specified to account for passing either A or B.

2, to indicate that the 4974 wire image buffer is to be initialized with the standard 64-character EBCDIC set. If the count operand is zero, no data is transferred. If the count is 8 or less, each bit of the data string indicates replacement (1) or non-replacement (0) of the corresponding character in the standard set with the alternate character as defined in the attachment. If 2 is specified, function must be PUTSTORE.

Example 1: Initialize a 4974 wire image buffer

```
          TERMCTRL  PUTSTORE,*,*,0,TYPE=2
```

Example 2: Initialize the 4974 wire image buffer to the standard EBCDIC character set and replace the standard dollar sign ($) with its alternate, the English sterling symbol (hex code 5B) and the standard cent sign (¢) with its alternate, dollar sign ($), (hex code 4A).

```
          TERMCRTL  PUTSTORE,REPLACE,*,2,TYPE=2
 REPLACE  DATA      X'1200'
```

4978 Display

Syntax

```
label       TERMCTRL  function,op1,op2,count,TYPE=,ATTN=

Required:   function
Defaults:   none
Indexable:  op1,op2
```

Operands	Description	
function:		
	BLANK	Inhibits display of the contents of the 4978 screen. The contents of the internal buffer remain unchanged. If specified, no other operands are required.
	DISPLAY	Causes the screen contents to be displayed if previously blanked by the BLANK function. Any buffered output is also displayed and the cursor position is updated accordingly.
	TONE	Causes the audible alarm to be sounded if the audible alarm is installed.
	BLINK	Sets the cursor to the blinking state.
	UNBLINK	Sets the cursor to the non-blinking state.
	LOCK	Locks the keyboard.
	UNLOCK	Unlocks the keyboard.

TEXT

Data Definition

TEXT is used to define a standard text message or text buffer. The characters are stored in EBCDIC or ASCII code. The PRINTEXT instruction may be used to print this message buffer on a terminal. READTEXT may be used to read a character string from a terminal into the TEXT buffer. A count field is maintained as part of the TEXT buffer and indicates the number of characters in the message received or to be printed. The contents of the buffer will be:

BYTE	CONTENT	
0	Length	
1	Count	
2	First byte of text	(addressed by 'label')

For a diagram of a buffer layout see Figure 16 on page 307.

Syntax

```
label        TEXT  'message',LENGTH=,CODE=

Required:    'message' or LENGTH=
Defaults:    CODE=E       EBCDIC is the standard internal
                          representation of all character
                          data
Indexable:   none
```

Operands	Description
label	Refers to the address of first byte of text. Used in GETEDIT, PUTEDIT, READTEXT, and PRINTEXT.
'message'	Any text string defined between apostrophes. If this operand is not coded, LENGTH must be coded and the buffer will be filled with EBCDIC spaces. The count field will equal the actual number of characters between apostrophes. If the LENGTH operand is not coded, and the count value is even, LENGTH=count. However, if the count value is odd, LENGTH=count+1.

Use two apostrophes to represent each printable apostrophe

The symbol '@' will cause a carriage return return or line feed to occur for nonstatic screen terminals only.

LENGTH=	Defines the maximum size (in bytes) of the text buffer. If this operand is not coded, 'message' must be coded and LENGTH equals the actual number of characters between apostrophes. The message is truncated if LENGTH is exceeded. The maximum value is 254. (Note that for $S1ASM the maximum length is 98 with a default of 64.) If 'message' is not coded, the text area will be initialized to EBCDIC blanks and the count byte will be equal to the length byte. If this operand is coded for a text buffer whose initial use will be for input, then the 'message' parameter should not be coded and the defined buffer will initially contain EBCDIC blanks.
CODE=	Defines the data type. Code E for EBCDIC, or A for ASCII. E is the default.

Example

```
MSG1     TEXT     'A MESSAGE'

MSG2     TEXT     'ABC',LENGTH=10,CODE=A

MSG#     TEXT     LENGTH=20
```

WAIT

Task Control

WAIT is used to wait for the occurrence of an event such as the completion of an I/O operation or a process interrupt. An event has an associated name specified by you. The initial status of any event defined by you is "event occurred" unless you explicitly reset the event with the RESET instruction before issuing the WAIT or reset the event in the WAIT instruction.

Syntax

```
label       WAIT  event,RESET,P1=

Required:   event
Defaults:   event not reset before wait
Indexable:  event
```

Operands	Description
event	The symbolic name of the event being waited upon. For process interrupt, use PIx, where x is a user process interrupt number in the range 1-99. For time intervals set by STIMER, use TIMER as the event name. Do not code RESET with TIMER. For disk I/O events, use DSn or the DSCB name from a DSCB statement as the event name. For terminals, use KEY to cause the task to wait for an operator to press the ENTER key or any PF key. Do not code RESET with KEY. Coding KEY with asynchronous supported terminals will give unpredictable results.
RESET	Reset (clear) the event before waiting. Using RESET will force the wait to occur even if the event has occurred and been posted complete. This parameter must not be specified when the WAIT is to be performed for the event specified in the EVENT operand of either a PROGRAM or a TASK statement.

P1=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

WAIT normally assumes the event is in the same partition as the currently executing program. However, it is possible to wait on an event in another partition using the cross-partition capability of WAIT. See the System Guide section on Cross-Partition Services.

When compiling programs with $S1ASM or the host assembler, ECBs are generated automatically by the POST instruction when needed. When using $EDXASM, ECBs must be explicity coded unless one of the system event names listed above is used.

When the WAIT is satisfied by way of a POST command, the post code is stored in both the event control block and in the waiting task's TCB code location.

WRITE

Disk/Tape I/O

Note: The Multiple Terminal Manager WRITE function is located in "WRITE" on page 381

WRITE is used to transfer one or more records from a storage buffer into a data set. For disk or diskette data sets you can write data either sequentially or randomly by relative record. The records are 256 bytes in length.

For tape data sets you can write data sequentially only. Tape records can be any length from 18 to 32767 bytes.

Syntax

```
label       WRITE   DSx,loc,count,relrecno|blksize,
                    END=,ERROR=,WAIT=, P2=,P3=,P4=

Required:   DSx,loc
Defaults:   count=1, relrecno=0 or blksize=256, WAIT=YES
Indexable:  loc, count, relrecno or blksize
```

Operands	Description
DSx	x specifies the relative data set number in a list of data sets defined by the user in the DS parameter of the PROGRAM statement. It must be in the range of 1 to n, where n is the number of data sets defined in the list. A DSCB name defined by a DSCB statement can be substituted for DSx.
loc	The symbolic name of the area from which data is to be transferred.
count	Specifies the number of contiguous records to be written. The maximum value for this field is 255. If you code 0 for this field, no I/O operation will be performed. A count of the actual number of records transferred will be returned in the second word of the task control block. If an end of data set condition occurs (fewer records remaining in the data set than specified by the count field) the

system will first write as many records as there is space remaining in a disk data set and then an end-of-data-set return code will be set.

relrecno — The number of the record, relative to the origin of the data set, which is to be written. Numbering begins with 1. This parameter may be either a constant or the label of the value to be used. A specification of 0 for relrecno indicates a sequential WRITE.

Sequential READs and WRITEs start with relative record one or the record number specified by a POINT instruction. The supervisor keeps track of sequential READs and WRITEs and increments an internal next record pointer for each record read or written in sequential mode (relrecno parameter is 0). Direct READs and WRITEs (relrecno parameter is not 0) may be intermixed with sequential operations, but these do not alter the next sequential record pointer used by sequential operations.

This disk WRITE operand cannot be used in the same instruction with the tape WRITE blksize operand.

blksize — This optional parameter specifies the size, in bytes, of the record to be written to a tape data set. The range is 18 to 32767 bytes. The value can be expressed as either a constant or as the label of the value to be used. If this operand is not coded, or if 0 is coded, the default value of 256 bytes is substituted.

This tape WRITE operand cannot be used in the same instruction with the disk WRITE relrecno operand.

END= — For disk or diskette, use this optional operand to specify the first instruction of the routine to be invoked if an end-of-data-set condition is detected (Return Code=10). If this operand is not specified, an EOD will be treated as an error. This operand must not be used if WAIT=NO is coded.

For tape, if an end-of-tape (EOT) condition is detected, the EOT path will be taken with return code 24, even though the block was successfully written. See the CONTROL statement for setting the proper end-of-data (EOD) indicators for an output tape. Multiple blocks (if specified by the count field) might not have been successfully written. The second word of the TCB contains the actual number of blocks written. This parameter is not valid

Return Codes

Code	Condition
-1	Successful
7	Link module in use
8	Unable to load $IAM
10	Invalid request
12	Data set shut down
13	Module not included in load module
22	Invalid IACB address

EXTRACT

Indexed Access Method

The EXTRACT request returns information from a File Control Block to the user's area. The FCB contains such things as the blocksize, key length, and data set and volume names of the indexed file. The FCBEQU copycode module contains a set of equates to map the File Control Block.

Syntax

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

Required:   iacb, buff
Defaults:   size = full FCB (or FCB extension)
            type = FCBNRM
Indexable: None
```

Operands	Description
iacb	The label of the word containing the IACB address returned by PROCESS or LOAD.
(buff)	The label of the user area into which the File Control Block (or FCB Extension) is copied. The area must be large enough to contain the requested portion of the FCB. Use the COPY statement to include FCBEQU in your program so the FCB and/or FCB Extension fields can be referenced by symbolic names.
(size)	The number of bytes of the FCB or FCB Extension to be copied into your buffer area. The size of the File Control Block is the value of the symbol FCBSIZE in the FCBEQU table. The actual size of the FCB Extension is the value of the symbol FCBXSIZ in the FCBEQU equate table. Either of these symbols can be coded for the size parameter.

(type) Type of extract to perform. Code one of the following:

<u>FCBNRM</u>	Extract File Control Block
<u>FCBEXT</u>	Extract File Control Block Extension

Note: FCBEXT is invalid for indexed data sets defined with a version of the Indexed Access Method prior to Version 1, Modification Level 2. (FCB Extension does not exist.)

Return Codes

Code	Condition
-1	Successful
7	Link module in use
8	Unable to load $IAM
12	Data set shutdown
13	Module not included in load module
22	Invalid IACB address
100	I/O error reading FCB extension
122	Invalid file for type FCBEXT

GET

Indexed Access Method

The GET request retrieves a single record from the indexed data set and places the record in a user area. The data set must be opened in the PROCESS mode.

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

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

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

<u>Syntax</u>

```
label       CALL      IAM,(GET),iacb,(buff),(key),
                     (mode/krel)

Required:   all
Defaults:   mode/krel=EQ
Indexable:  none
```

<u>Operands</u>	<u>Description</u>
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The label of the user area into which the requested record is placed.

(key) The label of your key area containing the key identifying the record to be retrieved and preceded by the lengths of the key and area. This area has the standard TEXT format and may be declared using the TEXT statement. This format is as follows:

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

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

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

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

Return Codes

Code	Condition
-1	Successful
-58	Record not found
-80	End of data
7	Link module in use
8	Unable to load $IAM
10	Invalid request
12	Data set shut down
13	Module not included in load module
14	Invalid index block found
22	Invalid IACB address
100	Read error
101	Write error

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

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

Return Codes

Code	Condition
-1	Successful
-58	Record not found
-80	End of data
7	Link module in use
8	Unable to load $IAM
10	Invalid request
12	Data set shut down
13	Module not included in load module
14	Invalid index block found
22	Invalid IACB address
100	Read error
101	Write error

LOAD

Indexed Access Method

Note: Task control LOAD is located under "LOAD" on page 194.

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

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

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

Syntax

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

Required:    all
Defaults:    mode=(SHARE)
Indexable:   none
```

Operands	Description
iacb	The label of a 1-word variable into which the address of the indexed access control block (IACB) is returned.
(dscb)	The name of a valid DSCB. This name is DSn, where n is a number from 1 - 9, corresponding to a data set defined by the PROGRAM statement. It can also be a name supplied by a DSCB statement. In the latter case you must have previously opened the DSCB with either the $DISKUT3 utility or with a DSOPEN statement.

Return Codes

Code	Condition
-1	Successful
7	Link module in use
8	Unable to load $IAM
10	Invalid request
12	Data set shut down
13	Module not included in load module
14	Invalid index block found
22	Invalid IACB address
60	Out of sequence or duplicate key (LOAD only)
61	End of file
62	Duplicate key found (PROCESS only)
70	No space for insert
100	Read error
101	Write error

PUTDE

Indexed Access Method

PUTDE deletes a record from an indexed data set. The record must have been previously retrieved by a GET or GETSEQ in update mode. Deleting the record creates free space in the data set. The PUTDE releases the lock placed on the record by the GET or GETSEQ.

Syntax

```
label       CALL      IAM,(PUTDE),iacb,(buff)


Required:   all
Defaults:   none
Indexable:  none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The name of the area containing the record previously retrieved by GET or GETSEQ.

Return Codes

Code	Condition
-1	Successful
7	Link module in use
8	Unable to load $IAM
10	Invalid request
12	Data set shut down
13	Module not included in load module
14	Invalid index block found
22	Invalid IACB address
85	Key was modified by user
100	Read error
101	Write error

PUTUP

Indexed Access Method

The record in your buffer (buff) replaces the record in the data set. The record must have been retrieved by a GET or GETSEQ in update mode. You must not change the key field in the record or the contents of the key variable in the GET request. The Indexed Access Method checks for and prohibits key modification. The PUTUP releases the lock placed on the record by the GET or GETSEQ.

Syntax

```
label       CALL      IAM,(PUTUP),iacb,(buff)


Required:  all
Defaults:  none
Indexable: none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(buff)	The label of the user area containing the record to replace the one previously retrieved.

```
ENTER CIRCBUFF ENTRY POINT: 62EE


MACHINE/PROGRAM CHECK STATUS REPORT

SINCE IPL     10 STATUS ENTRIES HAVE BEEN RECORDED

S/EAK TCBA  PSW  SAR  IAR  AKR  LSR   0    1    2    3    4    5    6    7

 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
 0100 0852 0802 0000 0000 0000 88D0 6E30 6E54 7352 6DFA 6E58 8023 0046 0000
 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
 0100 0852 0802 0000 0000 0000 88D0 6E30 6E54 7352 6DFA 6E58 8023 0046 0000
 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
 0100 0138 8002 6C31 1E6A 0000 88D0 6C30 6B7E 6C38 6C31 6C32 005C 00B8 0000
```

Figure 20. Format and Display Trace Data: This figure shows the result of the preceding program.

EXAMPLE 14: USE OF INDEXED ACCESS METHOD

This program gives an example for each of the Indexed Access Method function calls. The indexed data set is opened first in LOAD mode and ten base records are loaded followed by a DISCONNECT. Next the same data set is opened for processing. A GET request is performed for the first record whose key is greater than 'JONES PW'. Two more records are retrieved sequentially and then the ENDSEQ call releases the file from sequential mode. A record is then retrieved directly by key and updated. Another record is retrieved sequentially and deleted. A new record is inserted and another one is deleted by their unique keys. Finally, an example of extracting information from the file control block is shown. Upon successful completion the message "Verification Complete" will be displayed upon the console. This program requires that an Indexed Access Method data set has been defined with the $IAMUT1 utility according to the following specifications:

BASEREC	10
BLKSIZE	256
RECSIZE	80
KEYSIZE	28
KEYPOS	1
FREEREC	1
FREEBLK	10
RSVBLK	0
RSVIX	0
FPOOL	0
DELTHR	0
DYN	0

INDEXED ACCESS METHOD

```
Instruction  Operands

CALL          IAM,(DELETE),iacb,(key)

CALL          IAM,(DISCONN),iacb

CALL          IAM,(ENDSEQ),iacb

CALL          IAM,(EXTRACT),iacb,(buff-addr) [,(size
|             FULL|byte-value)] [,(type-FCBNRM | FCBEXT)]

CALL          IAM,(GET),iacb,(buff-addr),(key)[,(SHARE)|
              (EXCLUSV)/(EQ)|(GT)|(GE)|(UPEQ)|(UPGT)|(UPGE)]

CALL          IAM,(GETSEQ),iacb,(buff-addr),(key)[(SHARE)|
              (EXCLUSV)/(EQ)|(GT)|(GE)|(UPEQ)|(UPGT)|(UPGE)]

CALL          IAM,(LOAD),iacb,(dscb-addr|DSn),(opentab-addr)
              [,(SHARE)|(EXCLUSV)]

CALL          IAM,(PROCESS)(dscb-addr),(opentab-addr)
              [,(SHARE)|(EXCLUSV)]

CALL          IAM,(PUT),iacb,(buff-addr)

CALL          IAM,(PUTDEL),iacb

CALL          IAM,(PUTUP),iacb,(buff-addr)

CALL          IAM,(RELEASE),iacb
```

Instruction	Operands
CALL	ACTION[,(buffer-addr),(length),(crlf addr)]
CALL	BEEP
CALL	CDATA,(type),(userid),(userclass), (termname),(buffersize)
CALL	CHGPAN
CALL	CYCLE
CALL	FAN
CALL	FILEIO,(FCA-addr),(buffer-addr),(return-code-addr)
CALL	FTAB,(table),(size),(return-code-addr)
CALL	LINK,(pgmname)
CALL	LINKON,(pgmname)
CALL	MENU
CALL	SETCUR,(row-addr),(column-addr)
CALL	SETPAN,(dsname-addr),(return-code-addr)
CALL	WRITE,(buffer-addr),(length),(crlf addr)