IBM

Series/1

SC34-0314-2

File No. S1-35

LICENSED
PROGRAM

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

IBM

Series/1

SC34-0314-2

File No. S1-35

LICENSED
PROGRAM

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

Third Edition (April 1980)

Use this publication only for the purpose stated.

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

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services which are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

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

This publication could contain technical inaccuracies or typographical errors. A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Systems Publications, Department 27T, P.O. Box 1328, Boca Raton, Florida 33432. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

SUMMARY OF AMENDMENTS

New Instructions

In Chapter 3 the CONTROL instruction has been added to support the IBM Series/1 4969 Magnetic Tape Subsystem

Instruction and Statement List

- "Appendix A" has been added to list all of the Event Driven Language statements and instructions with their available operands and default values.

Modified Instructions

The following instructions and statements have been modified to include support for the IBM Series/1 4969 Magnetic Tape Subsystem:

- DSCB
- POINT
- PROGRAM
- READ
- WRITE

Summary of Amendments continued

Bibliography

The Bibliography lists the books in the EDX library and a recommended reading sequence. Other publications related to EDX are also listed.

Miscellaneous Changes

This manual has been modified to include new function and to improve technical accuracy and clarity. New material and technical changes are indicated by vertical bars in the left margin.

HOW TO USE THIS BOOK

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

PURPOSE

The Language Reference contains all details for coding individual Event Driven Language (EDL) instructions, except those used exclusively for remote communications and advanced terminal applications. Examples in the book illustrate the use of many EDL instructions in different applications.

AUDIENCE

The Language Reference is intended for application programmers who write and maintain application programs using EDL. The programmer is expected to know the Event Driven Language. EDL can be learned by using the IBM Series/1 Event Driven Executive Event Driven Executive Study Guide, SR30-0436, available through your local IBM Branch Office.

HOW THIS BOOK IS ORGANIZED

This manual is divided into six chapters and one appendix:

- "Chapter 1.Introduction" describes the Event Driven Language. It introduces each instruction or statement and describes its format. It also presents information about registers and parameter naming operands.

- "Chapter 2. Instructions and Statements - Overview" contains the instructions divided into categories according to their general use. These categories are arranged in alphabetical order.

- "Chapter 3. Instructions and Statements - Descriptions" contains a detailed description of each instruction or statement in the Event Driven Language, showing syntax rules, operands, and defaults. Each page contains a name tab at the top of the page for easy reference.

- "Chapter 4. Indexed Access Method" explains how this function is invoked and gives a detailed description of each instruction used.

- "Chapter 5. Multiple Terminal Manager" explains how this function is invoked and gives a detailed description of each instruction used.

- "Chapter 6. Programming Examples" contains coded program examples that use Event Driven Language instructions. Some examples do not represent complete programs because they do not include such instructions as PROGRAM, ENDPROG, and END statements.

- "Appendix A. Instruction and Statement List" lists the EDL, Communications, Indexed Access Method, and Multiple Terminal Manager instructions and statements. The lists also include the operands, their value ranges, and default values. Once you become familiar with the instructions you can code most instructions directly from these lists.

EXAMPLES AND OTHER AIDS

Throughout this book, coding examples and illustrations are used to clarify coding techniques and requirements. Coding examples are fully executable portions of complete programs that can be entered as shown. Coding illustrations are non-executable portions of incomplete programs that show the correct format of all required parameters on a statement. Missing code, or code provided by you, is indicated by a series of three vertical or horizontal dots.

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

- A Summary of Amendments lists the significant changes made to this publication since the last edition

- A Bibliography:

  - Lists the books in the Event Driven Executive library along with a brief description of each book and a recommended reading sequence

  - Lists related publications and materials

- A Glossary defines terms

- A Common Index which includes entries from each book in the Event Driven Executive library

RELATED PUBLICATIONS

Related publications are listed in the bibliography.

SUBMITTING AN APAR

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

The Language Reference is written for programmers who write and maintain application programs in the Event Driven Language (EDL). You are expected to be familiar with the overview information in the System Guide.

The Event Driven Language is a programming language designed for coding application programs. The language is designed at a level that allows flexibility for the application programmer without sacrificing productivity and is efficient in execution. The language can be used effectively for virtually any type of application.

The Event Driven Language contains many advanced features which provide great flexibility in application programming. For example, it allows exiting to and returning from other programs or routines where this level of complexity is required. It provides automatic translation for reading and writing alphabetic, numeric, or alphameric data to and from graphic screens. The language provides different levels of control for I/O operations. You can use the Event Driven Language to program I/O and allow the program to be device independent in most cases or you can control I/O devices at the machine instruction level.

An application program consists of instructions combined to form a task. A program can consist of one or more tasks. Each task has an assigned priority which is used by the supervisor to allocate system resources for task execution.

Application programs or tasks are made up of Event Driven Language instructions ~~that~~ have been processed by a compiler or assembler and prepared for execution by the $UPDATE/LINK system utilities. At execution time, the Event Driven Executive (EDX) Supervisor/Emulator analyzes the compiled or assembled format of an instruction and links to the appropriate supervisor routine to perform the operation. Following the completion of each instruction, the supervisor processes the next sequential instruction in the highest priority task that is in a ready state.

Programs written using the statements in this manual can be processed by any one of the following:

- Event Driven Language compiler $EDXASM (5719-XX2 or 5719-XX3)
- Macro Assembler, $S1ASM (5719-ASA), in conjunction with the macro library of program number 5719-LM5 or 5719-LM6

- S/370 Program Preparation Facility (5798-NNQ) which will be referred to as the host assembler in conjunction with the macro library of 5740-LM2 or 5740-LM3

Note: Throughout this manual, the S/370 facility is referred to as the host assembler.

LAYOUT AND STRUCTURE OF EDL PROGRAMS

There are three basic components in an Event Dirven Executive application:

- The Series/1 machine configuration definition
- The application I/O definitions
- The instructions and data areas comprising the application program

This three-part division minimizes the dependence of the application program on a particular system hardware configuration. In addition, the sensor based I/O definitions are checked against the machine configuration to reduce the execution time errors resulting from incorrect I/O assignments.

The "System Configuration" section of the System Guide describes the statements which define the hardware features on the Series/1. There are many optional components in the Event Driven Executive supervisor; their selection depends upon the configuration of the Series/1 for which the supervisor is compiled or assembled. A set of configuration statements beginning with SYSTEM are used to compile the configuration data which is then stored with the supervisor during installation.

The I/O devices and data sets used by an application are defined in the program itself. The PROGRAM statement must be the first statement in every EDL program. Operands on the PROGRAM statement and several I/O definition statements are provided to specify the symbolic device names, data set names, options, techniques and defaults to be used by the program. These optional statements are normally grouped together immediately following the PROGRAM statement Every program is automatically provided with a default definition of one terminal, the terminal from which the program was invoked. Up to 9 data sets can be made available for use simply by identifying them with the DS operand of PROGRAM. Many applications require no additional I/O descriptions.

The balance of an application program consists of its logic, data manipulations, I/O requests, and data. Because the Event Driven Language is both simple and powerful, it often requires very few instructions to describe a complete application program.

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.

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.
- The following labels are reserved for system use:

- All labels beginning with a $

- R0, R1, R2, R3, R4, R5, R6, R7, FR0, FR1, FR2, FR3

- #1, #2

- RETURN (except when used in the instruction to end a user exit routine)

- SETBUSY

- SUPEXIT

- SVC

• The operands, opnd1,opnd2,...opndn, are labels, names, or values defined for each instruction. Operands may also take the form NAME=name. This is called a "keyword" operand. Within any one instruction, the total positional and keyword operands must not exceed 50.

  The parameter naming operands, P1=,P2=,...Pn=, or P=(...) are used to allow modification of instruction parameters at execution time. This is discussed further on the following pages.

Instruction formats are illustrated in the following example of a simple program with a primary task ADDTEN. The first statement, PROGRAM, starts the program and defines the entry point as DOTEN. The DATA statement defines the variable COUNT to be 0. The first instruction has the label DOTEN, which starts a DO loop with a count of 10. The next instruction adds 1 to a variable, COUNT, which was initialized to 0 by the DATA statement. The ENDDO specifies the end of a DO loop. The ADD instruction is executed 10 times, then PRINTEXT and PRINTNUM instructions print the result on a terminal. The PROGSTOP statement terminates the program execution. The ENDPROG and END statements must be the last two statements of an Event Driven Executive application source program.

```
ADDTEN   PROGRAM    DOTEN

COUNT    DATA       F'0'       INITIALIZE COUNT TO 0

DOTEN    DO         10,TIMES   LOOP 10 TIMES
           ADD      COUNT,1    INCREMENT COUNT BY 1
         ENDDO
         ENQT
         PRINTEXT   'RESULT ='
         PRINTNUM   COUNT
         DEQT
         PROGSTOP

         ENDPROG
         END
```

The message will be: RESULT=10. This will be displayed on the terminal invoking this program.

Note: The program examples, starting in "Chapter 6. Programming Examples" on page 383 can be of great assistance in understanding the usage of many of the instructions introduced here and described in detail beginning in "Chapter 3. Instruction and Statement Descriptions" on page 51.

ADDRESS INDEXING FEATURE

Two software registers are available to you for each task and may be referenced in many instructions to provide indexed addressing. The registers themselves are referenced by the names #1 and #2. Except where specifically prohibited, the registers may be used in the same manner as any other variable in the program. For example, the integer arithmetic, logical, data movement, and program sequencing instructions may be used to set, modify, and test these registers. Other instructions are only permitted to use these index registers in the parameter format (parameter,#r). For example, the instruction

```
          MOVE     #1,0
```

will set register #1 to the value 0. The instruction

```
          MOVE     #2,A
```

will set register #2 to the contents of the variable A. An example of the use of the register as the from parameter is:

```
          ADD      A,#1
```

Here, the contents of register #1 will be added to the variable A and the result will be placed in A. It may be necessary to set a register to the address of a variable or vector. This is accomplished with the MOVEA instruction. For example,

```
          MOVEA    #2,BUFFER1
```

sets register #2 to the address of the variable BUFFER1.

The syntax of an instruction parameter in which an index register is specified is in the form:

```
  (parameter,#r)
```

where parameter is either an address or a constant and r is either a 1 or a 2. The effective address will result from the sum of the address (or constant) specified by parameter and the current contents (constant or address) of the referenced index register. Only one of the variables, either the parameter or the index register, can be an address; the other must be a displacement constant.

For example, if #1 = 2 then the indexed instruction

```
       MOVE    A,(B,#1)
```

would be equivalent to the nonindexed instruction

```
       MOVE    A,B+2
```

as would

```
       MOVE    A,(2,#1)
```

if register #1 contained the address of B. The following example illustrates the use of the indexing feature in a DO loop to find a value of -350 in a vector containing 1000 elements:

```
            MOVE    #1,0
            DO      1000,TIMES
              IF    ((BUF,#1),EQ,-350),GOTO,FOUND
              ADD   #1,2
            ENDDO
              .
         did not find a match
              .
  FOUND     MOVE    DISP,#1
              .
              .
```

The index register, #1, is set to 0, a DO loop is started to execute 1,000 times. The buffer BUF has an implied length of 1,000 words (2,000 bytes). A test is made on the first value of the buffer, and if a match occurs, a branch to the label FOUND is made. If not, the register is incremented by 2 (2 bytes = 1 word) and the second value tested, and so on. When the value -350 is found in the buffer, the displacement from the start of the buffer, which is now contained in #1, is saved at the location DISP.

Each task has its own #1 and #2 index registers and the supervisor always interprets instructions using the currently executing task's registers. Thus, individual programs and individual task within the same program will have different values in their respective index registers. If a subroutine is called by several different tasks, it uses the calling task's #1 and #2. Overlays, however, are independent programs with their own tasks and therefore have their own registers and do not use the invoking task's registers. Also, when moving data into or out of #1 or #2 with the cross-partition facility of MOVE, remember that the index registers are in the executing programs partition.

USE OF THE PARAMETER NAMING OPERANDS (PX=)

In some programs it is necessary to complete the parameters used in certain instructions during execution. The Px operands permit this to be done easily. The Px operands refer to other operands within the same instruction in the following manner: P1 refers to opnd1, P2 refers to opnd2, and so on, through the instruction according to the syntax for each instruction. For example, the number of times to execute a loop may not be known at compile time. You may assign a name to a parameter by adding the keyword Px=NAME to the instruction definition, where x is the operand number (1,2,..). The operand number specified in the Px keyword is given the name specified by the Px operand. This name can then be used as an operand in other instructions that modify the parameter at execution time. The following example shows a typical use of a Px operand. The P1=M operand on the ADD instruction causes the label M to be placed on the first operand in the ADD parameter list. The parameter list is shown as DC instructions; these are automatically generated by the compiler. The MOVEA instruction (prior to the ADD) uses the label M to modify the variable to be used by the ADD instruction.

```
             MOVEA    M,NAME      address of name
             .
             .
             ADD      A,B,P1=M
     +       DC       A($ADD)     ADD operation
     +M      DC       A(A)        parameter 1
     +       DC       A(B)        parameter 2
             .
             .
```

Execution of the MOVEA instruction changes the contents of the first operand of the ADD instruction from:

```
     +M      DC       A(A)
```

to:

```
     +M      DC       A(NAME)
```

and execution of the ADD instruction would result in the addition of the contents of NAME and B.

TASK CODE WORDS

Each task in the Event Driven Executive environment has a task control block (TCB) associated with it. The first two words of the TCB are called task code words and can be accessed using the taskname. The taskname is described more fully in "Chapter 3. Instruction and Statement Descriptions" on page 51 under the statements PROGRAM and TASK.

The first task code word (word 0) is used by the EDX supervisor to store the return code of various instructions. This word can be tested to determine the value of the return code of those instructions that return a code following their execution. This test must be performed immediately after the instruction execution because the task code word may be overlaid by the return code of the next instruction.

The second task code word (word 1) may contain additional information unique to the function being used or the condition encountered.

SYMBOLIC SENSOR BASED I/O ASSIGNMENTS

The sensor-based I/O instruction (SBIO) refers to the I/O devices using a 3- or 4-character name. The first 2 characters identify the type of device: AI, DI, PI, AO, and DO for analog input, digital input, process interrupt, analog output, and digital output, respectively. The next 1 or 2 characters are the user identification for the device, a number between 1 and 99. For example, the user may have three analog input terminals assigned to him. He identifies them as AI1, AI2, and AI3. The assignment of the actual physical addresses is done prior to compiling the application program using the sensor based I/O definition statement (IODEF). Therefore, all SBIO instructions become independent of the physical location of the sensor I/O points.

The assignment of sensor I/O symbolic addresses is described under "IODEF" on page 185. Figure 1 on page 10 depicts the relationship between symbolic I/O, IODEF, and the physical I/O unit.

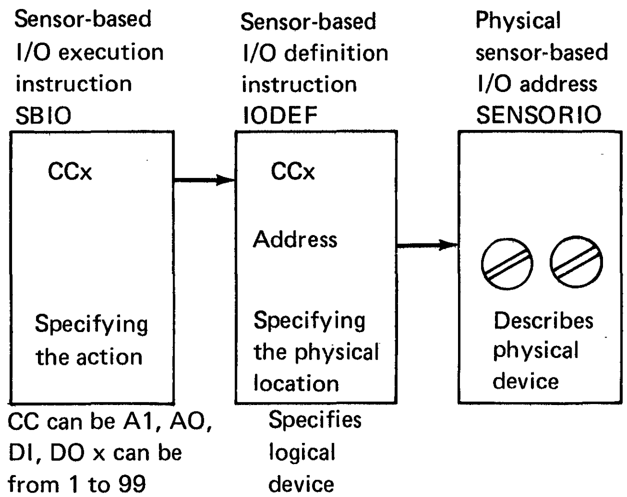


Figure 1. Symbolic I/O Assignment

SYMBOLIC TERMINAL I/O ASSIGNMENTS

Symbolic addressing is also used for terminal devices. In the application the terminal is identified with a name which at execution time is related to the TERMINAL system configuration statement with a label of the same name. A default terminal can be accessed by omitting the terminal name from the terminal I/O statements in the application. This causes the terminal which invoked the application to be used for the I/O and makes the application completely independent of terminal addresses.

SYMBOLIC DISK/TAPE I/O ASSIGNMENTS

Symbolic addressing for disk, diskette, or tape devices is achieved by having all I/O statements in the application refer to the symbolic data set control block DSCB name. At execution time, the data set and volume defined by the DSCB are found, and I/O is directed to the proper physical device addresses. If desired, the data set and volume names can be supplied by you at the terminal when the program is loaded for execution.

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) 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.

PROGEQU

The PROGEQU equate table provides maps of the program header (built by the PROGRAM statement) and the supervisor's communication vector table (CVT).

TCBEQU

The TCBEQU equate table provides a map of the task control block (TCB) built by either the TASK or PROGRAM statements.

TDBEQU

The TDBEQU equate table provides a map of the tape data block (TDB) built by the TAPE system configuration statement.

CHAPTER 2. INSTRUCTIONS AND STATEMENTS - OVERVIEW

This chapter presents the coding instructions and statements grouped by functions and their usage and listed in alphabetical order according to function. For example, the WRITE instruction falls into the application type listed under "Disk/Diskette I/O Instructions" on page 22 and also repeated under "Tape I/O Instructions" on page 40. There are programming considerations with each group of instructions and you should be familiar with these considerations prior to coding the individual instructions.

Some instructions/instruction groups require the support of optional features in your hardware configuration. Before these features are accessible by your programs, various supervisor modules must be included in $LNKCNTL during your system generation. Refer to the System Guide for supervisor modules required for optional features support.

For detailed descriptions of individual instructions see "Chapter 3. Instruction and Statement Descriptions" on page 51 of this manual.

Binary Synchronous Communication Instructions

BSCCLOSE
BSCIOCB
BSCOPEN
BSCREAD
BSCWRITE

Binary synchronous communication instructions allow you to read and write data to a host system in binary synchronous mode. These instructions are described in detail in the Communications and Terminal Applications Guide.

Host Communications Facility Instructions (TP)

TP CLOSE	TP RELEASE
TP FETCH	TP SET
TP OPENIN	TP SUBMIT
TP OPENOUT	TP TIMEDATE
TP READ	TP WRITE

The TP instruction provides services used to communicate with the Host Communications Facility installed user program (IUP) on a S/370 processor. Detailed descriptions are described for these instructions in the Communications and Terminal Applications Guide.

DATA DEFINITION STATEMENTS

BUFFER	EQU
DATA	STATUS
DC	TEXT

Use the data definition statements to define storage areas and the data initially placed in these areas. The DATA and DC statements perform the same function and have the same operands. The Series/1 and host macro assemblers provide some additional operands for DC, but all operands shown in the DATA/DC description are accepted by both macro assemblers and $EDXASM unless otherwise noted.

DATA FORMATTING INSTRUCTIONS

CONVTD
CONVTB
FORMAT
GETEDIT
PUTEDIT

The data formatting instructions allows you to prepare formatted data for display on the terminals or printers attached to the Series/1. In addition, you can format data in storage and then allow the program to determine the destination.

The data formatting instructions FORMAT, GETEDIT, and PUTEDIT require that your object program be processed by the link edit program, $LINK, to include the formatting routines which are supplied as object modules. The EXTRN statements necessary to reference these modules are generated as part of the compilation of the instruction. The modules can be automatically included in your program when required by using the $LINK autocall facility and the $AUTO autocall list provided in ASMLIB. For information on the use of the AUTOCALL option of $LINK, refer to <u>Utilities, Operator Commands, Program Preparation, Messages and Codes</u>.

You may also build your own autocall list or include the format modules yourself. The modules names are:

$GPLIST	$PUAC
$GEER	$PUFC
$GESC	$PUIC
$GEAC	$PUXC
$GEFC	$PUHC
$GEIC	$PUSC
$GEXC	$PUEC
$GEPM	

DATA MANIPULATION INSTRUCTIONS

ADD	FDIVD	MOVEA
ADDV	FMULT	MULTIPLY
AND	FPCONV	SHIFTL
DIVIDE	FSUB	SHIFTR
EOR	IOR	SQRT
FADD	MOVE	SUBTRACT

Vector Data Manipulation

A vector is defined in this manual as a series of contiguous data elements; bytes, words, or double words. Operand 1 determines the beginning location of a vector and the count value determines the vector length. Operand 2 is applied to each element of the vector.

The ADDV and MOVE instructions are exceptions to this because they establish 2 vectors: operand 1 and operand 2 along with the count value. In these cases the first element of operand 2 is applied to the first element of operand 1, then the second element of operand 2 is applied to the second element of operand 1, and so on, until the count is exhausted.

If the MOVE instruction operand 2 is immediate data, an explicit constant, then only operand 1 is a vector.

Integer And Logical Instructions

Integer arithmetic, logical, and data movement operations are performed with instructions which have a common general form.

Data Representation

Arithmetic operands are interpreted as signed-binary integers with negative values represented in twos complement form. Single-precision operands consist of 16 bits including sign; double-precision operands consist of 32 bits including sign. Logical operands are interpreted as bit strings of the appropriate length: byte, word, or doubleword. Single- and double-precision operands of both types must be located on even address boundaries.

Overflow

Overflow conditions encountered during the integer instructions ADD, ADDV, SUBTRACT, and MULTIPLY are not reported by EDX.

Mixed-precision Operations

Allowable precision combinations for integer arithmetic 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	-
D	S	S	DSS	DIVIDE only

Legend: S = single precision
D = double precision

Operations Using Index Registers

Index registers may generally be treated as ordinary single-precision integer arithmetic or logical variables. However, results of a vector operation directed at the registers (#1 and #2) may not extend beyond #2.

Floating-Point Arithmetic Instructions

Floating-point arithmetic instructions share a common format. Attempts to perform floating-point operations on a Series/1 not equipped with the floating-point hardware result in a program check error. Floating-point support must also be included in the supervisor when it is generated. FLOAT=YES must be specified on both the PROGRAM and TASK statements whenever floating-point instructions are used in any task within a program.

Data Representation

Arithmetic operands are interpreted as signed floating-point numbers in either single- or extended-precision. Single-precision, for floating-point instructions, is 32 bits; double-precision is 64 bits. Further, the second data operand may be coded as an integer value between -32768 and +32767. This immediate data will be converted to a single precision floating point number prior to the arithmetic operation to be performed.

Operations Using Index Registers

The index registers (#1 and #2) cannot be used as operands in floating-point operations because the index registers are only 16 bits in size. These registers may be used to specify the address of a floating-point operand.

Return Codes

Floating-point operations produce return codes which are placed in the task code word. This word is referred to by taskname (see PROGRAM/TASK statements). These codes must be tested immediately after the floating-point instruction is executed or the code may be destroyed by subsequent instructions. The return codes are listed following the description of each individual floating-point instruction.

DISK/DISKETTE I/O INSTRUCTIONS

DSCB
NOTE
POINT
READ
WRITE

You are provided with both sequential and random access to disk or diskette data sets. When a program is first loaded for execution, all of your data sets have been opened for access (reading or writing) beginning with the first record. Sequential and random access operations may be intermixed. For instance, if five READ instructions, consisting of one record each have been initially issued to a data set, then the next sequential operation will normally take place with record number 6. A random access READ could be issued for some other record, say record 23, and the next sequential operation would still take place with record 6.

To open a data set during the execution of your program, you will need an OPEN subroutine. (For details on the OPEN subroutine, see "DSOPEN SUBROUTINE" in the System Guide).

Definitions For Disk Data Sets

Record: The basic unit of direct access storage available to an application program is a record on disk or diskette which contains 256 bytes of data. Records are contained in data sets, or may be free space in a library. Data set record numbering begins with 1.

Data Set: A data set is a group of reserved contiguous records which have have been assigned collectively a data set name consisting of 1 to 8 bytes. No special restrictions exist within the system for valid names, but the use of standard system utility programs for data set access and allocation dictates that an alphameric character string be used as a name. Data sets are contained in libraries.

A data set can contain either data or an executable program. The term member (of the library) is sometimes used when referring to either type of data set. These data sets can be further subdivided with the use of the $PDS utility which can partition an Event Driven Executive data set. $PDS uses the term members to describe a group of contiguous records within the partitioned data set which have been assigned a name.

Volume/Library: A library is a set of contiguous records which contains (1) a directory and either or both of the following: (2) a set of allocated data sets, (3) space available for the allocation of new data sets. A directory is a series of contiguous records which describe the library contents in terms of allocated data sets and free space. These records are at the beginning of the library. A library is contained in a volume.

A volume is a physical direct access storage device, or a subset thereof. Each volume is assigned a volume name of 1 to 6 alphameric characters. A volume begins on a cylinder boundary and contains an integral number of cylinders. The maximum volume size is 32,767 records. Only one volume can be placed on a diskette or in the fixed-head area of a disk, but disks may have as many volumes as disk storage will permit. Each volume can contain only one library.

Notes:

1. Additional information on direct access devices and organization can be found in the System Guide.

2. For each data set defined in a PROGRAM statement, a data set control block (DSCB) is generated in the program header. A DSCB is used to contain information about the current use of a data set within an active program such as the location of the data set and the next record number for sequential I/O. This allows the system to properly control access to the same data set by separate programs.

3. A DSCB is a serially reusable program resource; therefore, within a single program it is your responsibility to prevent simultaneous access to the same data set from separate tasks. It is recommended that access to a data set within a given program come from a single task. If, however, it is necessary in a given application to access the same data set from within different tasks in the same program the user should use ENQ and DEQ to ensure serialized use of the affected DSCB.

EXIO CONTROL INSTRUCTIONS

DCB
EXIO
EXOPEN
IDCB

I/O level control functions allow you to control, at a low level, any I/O device attached to the system. They give you the ability to control devices not otherwise available using Event Driven Language instructions. They also give you the ability to gain closer control of a device than is provided by other I/O facilities.

To use the EXIO control functions you should be familiar with I/O programming in assembler language. Refer to the section on Input/Output Operations in the manuals describing the processors for general descriptions of the immediate device control block (IDCB) and the device control block (DCB) and their use, and to the manuals describing the particular I/O device for specific information for that device.

You must be thoroughly familiar with the device to be controlled. The facilities provided by these instructions are approximately those provided by the Series/1 hardware for I/O. You must, by using EXIO instructions, explicitly control every aspect of the device's operations.

After you define each device to be controlled by an EXIODEV statement (see the System Guide), you can use the EXIO and EXOPEN instructions.

Each device must be controlled from one task at a time. Before a task relinquishes control of a device, it must assure that all interrupts from that device have been serviced.

You must not alter a DCB until the operation caused by the EXIO instruction which referenced it is complete. The IDCB may be modified after its use in an EXIO instruction.

I/O commands produced by the COMMAND operand of the IDCB statement are those used by IBM I/O devices and described in the publications which describe the processors and I/O devices. Any other device must be designed to respond to these same commands if these instructions are to be used to control it.

If an EXIO device produces interrupts, you must:

1. Open the device by executing an EXOPEN instruction. This allows the interrupt handler to return device information to the user's program.

2. Prepare the device by executing an EXIO instruction, so that it can interrupt the processor.

3. WAIT in one or more tasks for one or more ECBs which will be posted when an interrupt is received.

4. Obtain all information required to service the interrupt. This information is available from:

   a. Code word in ECB posted

   b. Interrupt identification word and level status register (see "EXOPEN" on page 129)

   c. Residual status (refer to the description of DVPARM4 operand statement in "DCB" on page 91)

   d. Cycle steal status (see description of listaddr operand of refid='exope', the EXOPEN instruction, and the description of COMMAND=SCSS operand of "IDCB" on page 175)

5. Prevent further interrupts if the interrupt servicing task is to terminate. This may be done by executing an EXIO instruction which specifies an IDCB with COMMAND=PREPARE and IBIT=OFF.

GRAPHICS INSTRUCTIONS

CONCAT	SCREEN
GIN	XYPLOT
PLOTGIN	YTPLOT

The graphics instructions provide a tool for the development of graphics applications. They can aid in the preparation of graphic messages, allow interactive input, and draw curves on a display terminal.

These instructions are only valid for ASCII terminals having a point-to-point vector graphics capability and compatible with the coordinate conversion algorithm for graphics mode control characters. This is described in detail in Internal Design. The function of the various ASCII control characters used by a terminal are described in the manual for that terminal. Such terminals may be connected to the Series/1 using the teletypewriter adapter.

When the Event Driven Executive instructions are used, detailed manipulation of terminal instructions and text messages is not required. All of the graphics instructions deal with ASCII data; therefore when an ASCII text string is sent to the terminal the XLATE=NO parameter should be coded.

There are six graphic instructions. They are used in the same manner as other instructions, except that the supporting code will be included in your program, rather than in the supervisor. If all instructions are coded in a program, this code requires approximately 1500 bytes of storage.

Use of the graphics instructions requires that your object program be processed by the link edit program, $LINK, in order to include the graphics functions which are supplied as object modules. EXTRN statements for the necessary modules are included in your program when the instructions are coded. The modules ($$CONCAT, $$SCREEN, $$YPLOT, $$GIN, and $$PGIN) can be automatically included in your program when required using the $LINK autocall facility. Use the $AUTO autocall list provided in ASMLIB for this purpose. Refer to Utilities, Operator Commands, Program Preparation, Messages and Codes for information on the use of the autocall option of $LINK.

For a list of terminals supported, see "Terminal Support" in the System Guide.

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.

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.

MULTIPLE TERMINAL MANAGER INSTRUCTIONS

ACTION	FAN	MENU
BEEP	FILEIO	SETCUR
CHGPAN	FTAB	SETPAN
CDATA	LINK	WRITE
CYCLE	LINKON	

The Multiple Terminal Manager is an optional licensed program which provides the Event Driven Executive user with a set of high-level functions designed to simplify the definition of transaction-oriented applications such as inquiry, file update, data collection, and order entry.

Transaction-oriented means that program execution is driven by terminal operator actions, typically, responses to prompts from the system. For example, a program executing under control of the Multiple Terminal Manager displays a menu screen offering the operator a choice of functions. Based on the operator's selection, the application program then performs processing operations such as reading information from a data file, displaying the data at the terminal, and waiting for the next response.

This prompt-response-process cycle between the Series/1 program and the terminal operator is the basic principle for the design of applications using the Multiple Terminal Manager.

The terminal manager simplifies such transactions by:

- Automatically allocating input and output buffers for the application program.

- Performing I/O operations to access fixed-screen formats from the screen file. The term screen in this discussion refers to the image which is displayed on the screen of an IBM 4979, 4978 or 3101 Display Station. Fixed-screen formats consist of unmodifiable text and definitions of possible areas for data input. On other systems, these may be referred to as maps, formats, or panels. Screens are built using the $IMAGE utility. (See <u>Utilities, Operator Commands, Program Preparation, Messages and Codes</u> for additional information.)

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

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

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

Multiple Terminal Manager application programs can be written in Event Driven Language, assembler language, COBOL, PL/I, or FORTRAN IV. Disk I/O can be performed using indexed-access or direct-access methods. Terminal support is provided for IBM 4979, 4978, and 3101 Display Stations and teletypewriter compatible terminals attached using the single line or multiline asynchronous communication adapters.

Note: Throughout the manual, when reference is made to the IBM 3101 Display Station, it is inferred to mean model 1 and model 2. However, model 2 is considered only in block mode (full screen).

The functions provided by the Multiple Terminal Manager are callable routines that perform terminal, disk and diskette input/output operations and control the execution of application programs.

The program-execution control and terminal I/O functions include:

- A routine (ACTION) to initiate the prompt-response terminal I/O cycle.
- A routine (CDATA) which provides information about the terminal which is controlling an executing program.
- Two routines (LINK and LINKON) to link to a new program from the currently executing program.
- A routine (MENU) to terminate program execution and return control to the Multiple Terminal Manager.
- A routine (CYCLE) to voluntarily give up control of the program area to other users. This allows a user controlled form of time sharing.

The Multiple Terminal Manager provides four callable functions for the specific control of the IBM 4978/4979 Display. They are:

- A routine (SETPAN) to retrieve a screen panel from disk and move it into the input and output buffers.
- A routine (SETCUR) to override the initial cursor position defined for that screen format.

- A routine (BEEP) to request the 4978 audible alarm be sounded on the next terminal I/O cycle.

- A routine (CHGPAN) to notify the terminal manager of changes to a screen before it is written.

For the teletypewriter user, the following functions are provided:

- A routine (ACTION) to write to the terminal and read a reply.

- A routine (WRITE) to write to the terminal without waiting for an operator response. Multiple writes may be used to write long messages, with the last message being written using ACTION.

- A routine (BEEP) to cause a bell character to be included in the next output line.

The FILEIO function provides the following for disk and diskette files:

- Automatic open of the requested file

- Indexed file support

- Direct file support

- Storage conservation through automatic open and close functions

Two programming aids are available using the Multiple Terminal Manager:

- A no-operation (FAN) adds programming compatibility with other programming environments.

- An unprotected field descriptor function (FTAB) describes the fields of the screen image in the input buffer.

The coding syntax for these functions are described in detail in "Chapter 5. Multiple Terminal Manager" on page 359 and are organized alphabetically by function name, such as ACTION, LINK, LINKON, and so on.

Use of these facilities requires the Multiple Terminal Manager Licensed Program (5719-MS1) and also the Indexed Access Method Licensed Program (5719-AM3) if indexed files will be used.

PROGRAM CONTROL STATEMENTS

CALL
CALLFORT
RETURN
SUBROUT
USER

Program control statements are used to define and control subsections within a program and can provide flexibility and save space. CALL, SUBROUT, and RETURN provide for the definition and use of a reusable section of code. Calling a subroutine and the returning to the mainstream program reduces repetition of code and program complexity.

CALL is also used to invoke the individual functions of the optional licensed programs Indexed Access Method and Multiple Terminal Manager.

The USER statement allows Event Driven Executive programs to utilize the Series/1 assembler language in those specialized cases where the Event Driven Language does not meet application requirements.

CALLFORT is used to invoke FORTRAN programs and subroutines.

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.

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)

The last form (n) provides a means for comparing data strings. For example, two 8-byte character strings may be compared or, similarly, two data buffers may be checked for equality. This form implies that both data1 and data2 are storage locations; an immediate second operand is not permitted.

Several forms of the IF and DO instructions are allowed. They are described in detail in the instruction descriptions in "Chapter 3. Instruction and Statement Descriptions" on page 51. The simplest form of the IF instruction is

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

If the word contained in the variable A is equal to the word contained in the variable B, the next sequential instruction will be executed. This is called the true portion of the IF-ELSE-ENDIF structure. For example:

```
  IF    (A,EQ,B)
     :
     :    (code for true condition)
  ELSE
     :
     :    (code for false condition)
  ENDIF
```

ELSE is an optional part of the structure, and if coded, the instructions following it are referred to as the false part of the structure. Therefore, in the example above, the instruction following the ELSE instruction will be executed if A is not equal to B. If ELSE is not coded, control passes to the instruction following the ENDIF if the condition is false.

The IF and DO instructions permit logically connected statements of the form:

```
  statement,OR,statement

  statement,AND,statement
```

More than two statements may be logically connected in an instruction. Logically connected statement strings are not evaluated according to normal Boolean reduction. Instead, the string is evaluated to be true or false by evaluating each sequence of:

```
  statement,conjunction
```

to be true or false as follows:

1. The expression is evaluated from left to right.

2. If the condition is true and the next conjunction is OR, or if there are no more conjunctions, the string is true and evaluation ceases.

3. If the condition is false and the next conjunction is OR, the next condition is checked.

4. If the condition is false and the next conjunction is AND, or if there are no more conjunctions, the string is false and evaluation ceases.

5. If the condition is true, and the next conjunction is AND, the next condition is checked.

The order of the statements and conjunctions in a statement string determines the evaluation of the string. It may be possible, by reordering the sequence of statements and conjunctions, to produce a statement string that will be evaluated to the same results as Boolean reduction of the statement. For example, the statement string

```
(A,EQ,B),AND,(C,GT,D),OR,(E,LT,F)
```

could be reordered as

```
(E,LT,F),OR,(A,EQ,B),AND,(C,GT,D)
```

without changing the results if evaluated by Boolean reduction. As a statement string in the IF or DO instructions, however, the two forms produce different evaluations. If A is not equal to B, but E is less than F, the first statement string will be evaluated false and evaluation will cease as soon as (A,EQ,B) is evaluated; however, the second statement string will be evaluated true if E is less than F, as would be expected from Boolean reduction for either the first or second statement string.

When writing code with structures, program readability is improved by indenting nested structures. Two spaces for each nesting level is recommended. For example:

```
IF    (A,EQ,B)
  :
  DO    WHILE,(X,NE,Y)
    :
    IF    (#1,EQ,1)
      :
    ENDIF
  ENDDO
ELSE
  :
ENDIF
```

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 oldest 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 most recent 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.

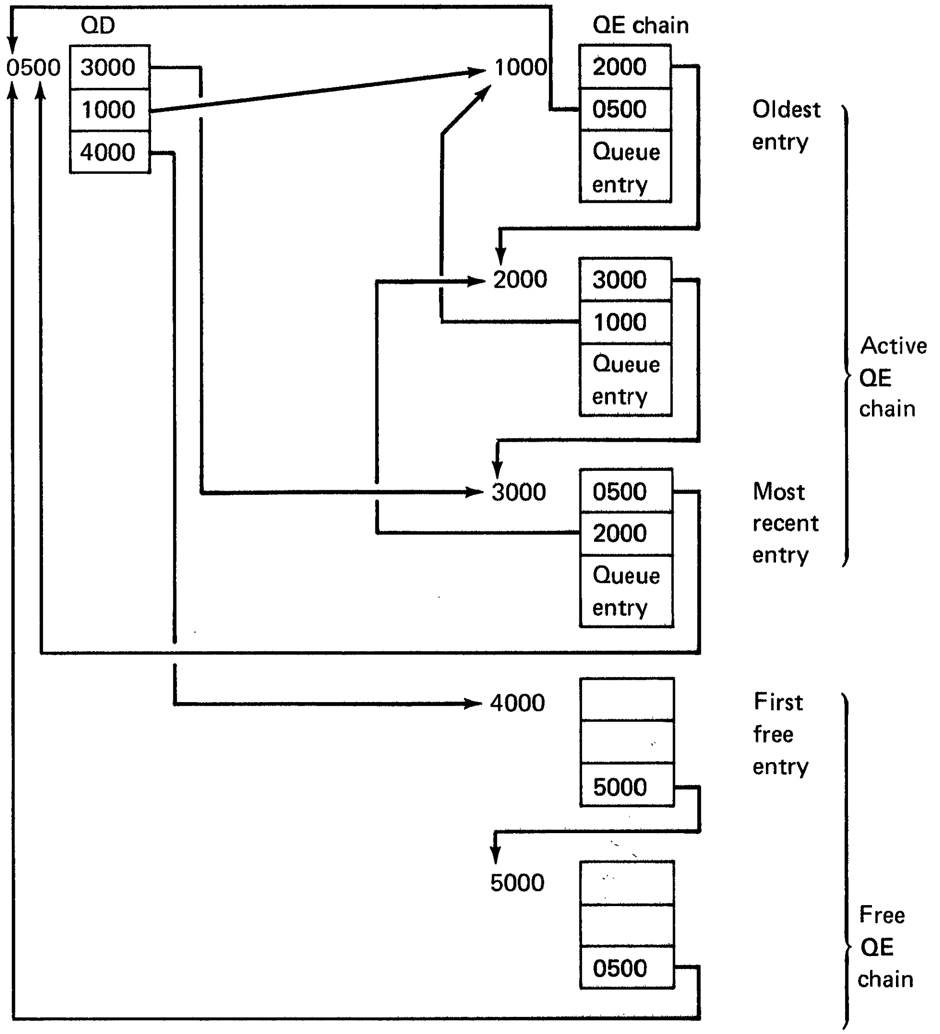


Figure 2. The Control Mechanism of Queue Processing

SENSOR-BASED I/O STATEMENTS

IODEF
SBIO
SPECPIRT

The sensor-based I/O statements provide the means for defining the devices, device addresses, and the general operating environment for the sensor-based application program. See Figure 1 on page 10 for a diagram showing the relationships.

The purpose of a sensor I/O application program is to communicate with sensor I/O units. This communication is used for monitoring or controlling a process outside the Series/1 processor from a program within the processor.

In sensor applications, a process produces either digital or analog signals. These signals are sensed by sensor devices and transferred through a sensor I/O unit to your sensor program. These signals can be compared to stored digital data for monitoring. For process control, the application program must write new values to the sensor units.

SYSTEM CONFIGURATION STATEMENTS

BSCLINE	HOSTCOMM	TAPE
DISK	SENSORIO	TERMINAL
EXIODEV	SYSTEM	TIMER

These statements are used only during the generation of a supervisor. For more information on System Configuration and a description of each statement, refer to the "System Configuration" topic in the <u>System Guide</u>.

TAPE I/O INSTRUCTIONS

CONTROL	POINT
DSCB	READ
NOTE	WRITE

These instructions control the IBM Series/1 4969 Magnetic Tape Subsystem and provide sequential access to magnetic tape data sets. When a program is first loaded for execution, all the data sets named in your PROGRAM statement have been opened for access (reading or writing) and are positioned to the first record.

Definitions For Tape Data Sets

Tape Label: A tape label consists of at least two 80-character records which describe the tape contents, such as date the tape was created, the block size and record length, and other pertinent data. This data is usually in a specific format and referred to as a standard label. Non-standard labels may be used but no automatic processing will be performed on such labels by EDX. There is also a trailer label which has a standard format and contains record count, block count, and so on for the tape. The use of labels is optional and if they are present they can either be processed or bypassed.

Record: The basic unit of tape data storage available to an application program is a record. A record may be any size between 18 and 32767 bytes. The default size of a record is 256 bytes.

File: A file is all the records between any beginning tape mark (TM) and an ending TM. The term file and data set are sometimes used interchangeably in tape record references, however, data set is the preferred term here.

Data Set: A tape data set is a set of consecutive records recorded on a magnetic tape. No special restrictions exist within the system for valid names, but the use of standard system utility programs for data set access and allocation dictates that an alphameric character string be used as a name.

A tape data set can only contain data, not executable code.

Volume: A volume is all of the records recorded on a reel of magnetic tape. Each volume is assigned a volume name of 1 to 6 alphameric characters.

Load Point: The beginning of tape (BOT) where the load point sticker is located. Normally this location is approximately 25 feet from the leading end of a reel of magnetic tape and placed on the glossy side of the tape near the front edge.

End of Tape (EOT): The EOT sticker which is located near the physical end of a reel of magnetic tape. During a WRITE or CONTROL WTM command, the tape drive sensing this sticker will raise the EOT condition in the tape drive causing a return code value of 24 to be returned. This sticker is normally far enough from the physical end of tape to allow a complete block of records to be written after it is sensed. It is located on the glossy side of tape near the rear edge.

Notes:

1. Additional information on magnetic tape devices and organization can be found in the System Guide.

2. For each data set defined in a PROGRAM statement, a data set control block (DSCB) is generated in the program header. A DSCB is used to contain information about the current usage of a data set within an active program such as the location of the data set and the next record number for sequential I/O. This allows the system to properly control access to the same data set by separate programs.

3. A DSCB is a serially reusable program resource; therefore, within a single program it is your responsibility to prevent simultaneous access to the same data set from separate tasks. It is recommended that access to a data set within a given program come from a single task. If, however, it is necessary to access the same data set from within different tasks in the same program, you should use ENQ and DEQ to ensure serial use of the affected DSCB.

4. A tape drive cannot be shared by multiple programs at the same time. You should not create or open multiple DSCBs for the same tape volume. If you pass a tape data set to another program (DS= operand of LOAD), the DSCB of the program issuing the LOAD will be disconnected from the tape data set to allow it to be passed to the program being loaded.

5. When passing DSCBs to overlay programs, it is suggested that the address of the DSCB in the root program be passed and not the data set itself. If the data set is passed, close offline (CLSOFF) will be invoked when the overlay terminates; when the overlay executes a PROGSTOP statement.

TASK CONTROL INSTRUCTIONS

ATTACH	ENDATTN	PROGSTOP
ATTNLIST	ENDTASK	QCB
DEQ	ENDPROG	RESET
DETACH	ENQ	TASK
ECB	LOAD	WAIT
END	POST	WHERES
	PROGRAM	

The basic unit of a program is a task. The PROGRAM statement defines the initial task. Many tasks may be active concurrently and asynchronously in a program. A task may be activated or attached, using the ATTACH command, by the primary task or by other tasks. Any combination of instructions may be used within a task and will be executed independently of other tasks. Tasks within a program may communicate with each other through common storage areas or through system instructions and event control blocks. The facilities of the Event Driven Executive supervisor provide the capability of synchronizing task execution.

A user-written application program is composed of one or more tasks. The instructions listed here are used to define tasks and to control which of the tasks are active at any given moment, plus other related functions. "Example 7: A Two Task Program With ATTNLIST" on page 395 and "Example 9: Floating Point, WAIT/POST, GETEDIT/PUTEDIT" on page 398 illustrate the use of several task-control instructions.

Several programs, each composed of one or more tasks, may be loaded from disk and run concurrently. When a user task gains control of the system, its instructions are executed until a higher priority task becomes ready, at which time the higher priority task gains control of the system.

A program may have more than one independently operating task and these tasks may communicate with one another using data storage locations or event control blocks within the specific program of which they are a part. Communication among tasks in separate programs can be accomplished using the cross-partition facilities provided with many of the task control instructions. Communication can also be accomplished using a user-provided common data storage area ($SYSCOM) in the supervisor. The services available for cross partition communication are described further in the System Guide under "Cross Partition Services."

It is your responsibility to write programs in such a way that the tasks operate in the desired sequence and terminate properly.

Concurrent execution of multiple tasks is shown in Figure 3 on page 43

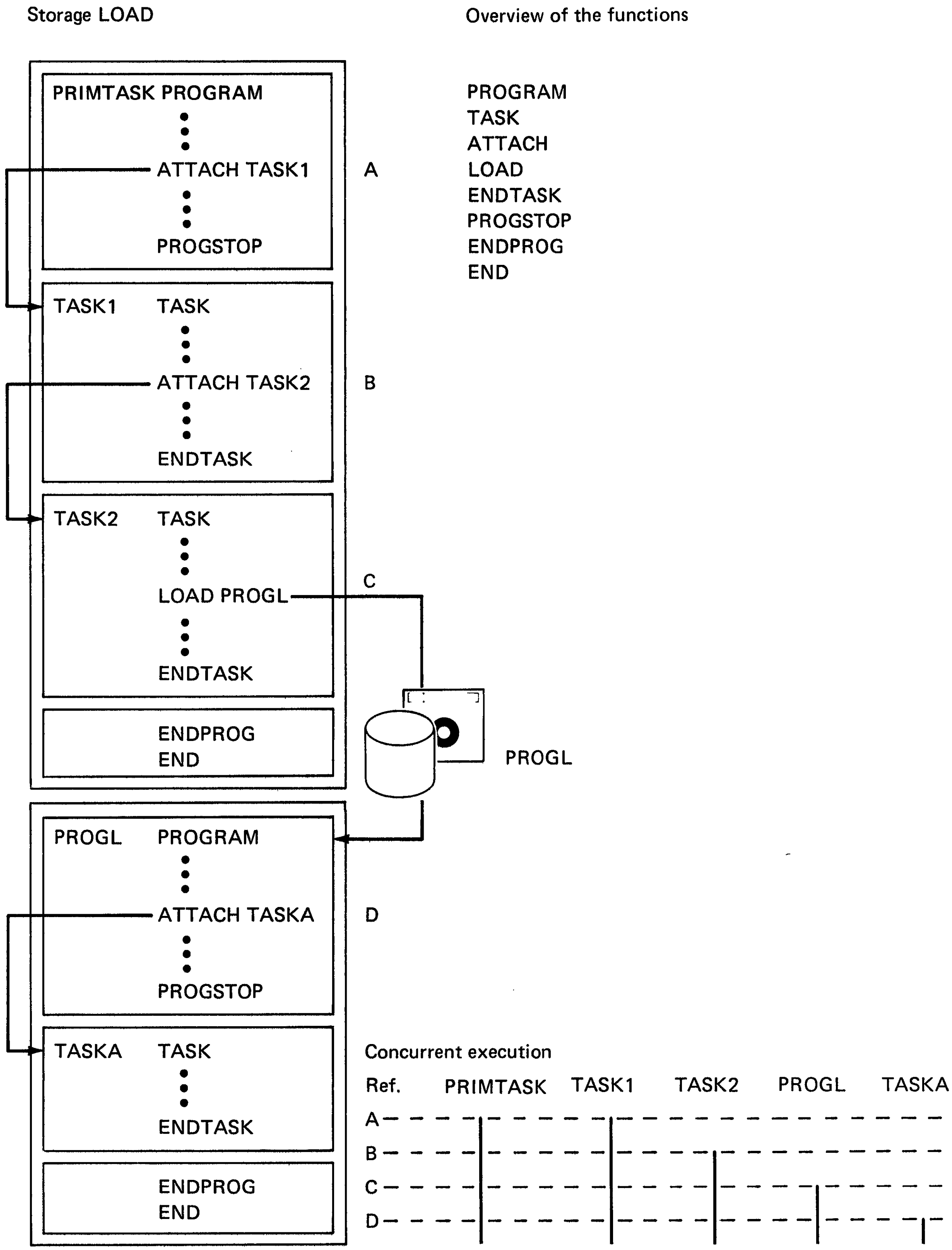


Figure 3. The Concurrent Execution of Multiple Tasks

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 may provide response to errors by means of the TERMERR operand 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 error 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, PRINTEXT, and TERMCTRL instructions in this manual. 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 operand is specified in a subsequent PRINTEXT statement. This information should be considered when coding a TERMERR routine.

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. Normally, when concatenated output exceeds the line-buffer capacity, subsequent output is lost until a new line indication is given; however, you can allow the generation of overflow lines by coding OVFLINE=YES in the TERMINAL statement for the device in question.

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 you to enter data in advance and thereby inhibit the associated prompting messages. Advance input is accomplished simply by entering more data on a line than may have been requested by the program. Subsequent input instructions which specify PROMPT=COND will then read data from the remainder of the buffered line, and will issue a prompting message only when the line has been exhausted. If you specify PROMPT=UNCOND with an input instruction, an associated prompting message is issued and the system waits for your input. The prompt message causes, as does every output message, cancellation of any outstanding advance input.

Attention Handling

Attention Keys: Program operation may be interrupted by pressing the keyboard ATTN key. When this key is recognized, the greater than symbol (>) is displayed and the operator may enter either a system function code (for example, $L) or a program function code defined by an active ATTNLIST. For ASCII terminals, the keys with character codes X'1B' (normally marked ESC on the keyboard) and X'7D' (normally the right brace) are both recognized as the attention key.

Program Function Keys: All program function keys on the IBM 4978/4979 Display Terminal are recognized by the attention list code $PF. In addition, individual keys may be separately recognized by $PF1 to $PF254. It is possible to provide separate entry points to the application code for particular keys, or for rapid response, a single entry for all keys. When the application program attention handler is entered for any program function key, the code for that key is placed in the second word of the keyboard task control block.

The order in which the program function key codes appear in the attention list is significant. For example:

```
ATTNLIST  ($PF1,ENT1,$PF5,ENT2,$PF,ENT3)
```

would cause the program to be entered at ENT3 for all program function keys except PF1 and PF5.

KEYBOARD AND ATTNLIST TASKS: When the ATTN key or one of the PF keys is pressed on a terminal, the keyboard task for that terminal gets control. Except for the hardcopy key (normally PF6), the PF keys are always matched against your ATTNLIST(s). For an ATTN, you enter a command which is first matched against the system ATTNLIST and then against your ATTNLIST(s). If the command matches the system ATTNLIST, appropriate system action is taken ($D, $L, etc.). If there is no match against any ATTNLIST, the message FUNCTION NOT DEFINED is displayed on the terminal. For a PF key or an ATTN command match against your ATTNLIST, the corresponding attention list task is given control. The appropriate application program attention routine then runs under this task. If the ATTNLIST task is already busy, the message, "> NOT ACKNOWLEDGED" is displayed on the terminal. You the have the option of reentering the command or pressing the PF key at a later time.

When the application program attention handler is entered, the index registers are initially set as follows:

#1 Address of task control block (TCB)
#2 Address of terminal control block (CCB)

The code for an interrupting key may therefore be obtained by coding, for example:

```
MOVE    CODE,(2,#1)
```

Screen Management

Support for the 4978/4979 display allows the application program to partition the screen into logical screens, and to manage a logical screen according to one of two basic modes, roll or static. The roll screen mode operates in a manner which simulates a typewriter terminal, while the static screen mode provides a convenient means for data display and data entry. The static screen mode is supplied only for the IBM 4978/4979 Display Terminals.

Roll Screens: Roll screens differ from typewriter printing media only in the absence of hardcopy and in the limited amount of display history which can be retained. The amount of history to be retained on a roll screen is specified through the NHIST parameter on the TERMINAL or IOCB statements. The value of this parameter defines the boundary between two areas of the screen, the history area (extending from TOPM to TOPM+NHIST-1), and the working area (extending from TOPM+NHIST to BOTM). The top of the working area is line 0 for purposes of forms control; the display proceeds from line 0 to the bottom margin, after which the working area is shifted into the history area, the working area is erased, and the display begins again at line 0.

Since screen shifting is implemented through a hardware mechanism which affects the entire physical screen line, shifting is not performed for roll screens whose left and right margins are other than 0 and 79. This protects adjacent logical screens from alteration. All other aspects of roll screen management are preserved.

Static Screens: The object of static screen management is to provide the application program with complete control over the screen image, and to allow the terminal operator to modify an 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. For the program function keys, the value is the integer corresponding to 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.

CHAPTER 3. INSTRUCTION AND STATEMENT DESCRIPTIONS

The Event Driven Language instructions and statements are presented here in alphabetic order. A brief description of the use of the instructions is provided where appropriate, followed by information on how to invoke any 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 standard Series/1 macro assembler format.

Each instruction is described in detail using the following format:

Instruction name

Functional description

Syntax

Operands

Coding examples

The "Address Indexing Feature" on page 6 can be used only with certain instructions and operands. The syntax description of each instruction specifies which operands, if any, are indexable.

The instructions are grouped by function beginning in "Chapter 2. Instructions and Statements - Overview" on page 15 and each functional group is presented alphabetically. Also, general information that is common to each group is discussed there.

You should note in this chapter that the functional group of each instruction is identified at the top of the first page of each instruction. You can use this functional identifier to refer back to the discussion in Chapter 2 of each functional group.

Some instructions are also shown in various programming examples beginning in "Chapter 6. Programming Examples" on page 383. These examples will give further assistance in the proper use of the more complex instructions.

ADD

Data Manipulation

The ADD instruction adds the signed value of operand 2 to the signed value of operand 1. The value of operand 2 remains unchanged.

| Note: An overflow condition is not indicated by EDX.

Syntax

```
label        ADD      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	The number of consecutive variables in opnd1 or RESULT upon which the operation is to be performed. The maximum value allowed is 32767.
RESULT=	The name of a variable or vector in which the result is placed. The variable specified by the first operand is not modified. This operand is optional.
PREC=XYZ	The precision value X applies to opnd1, Y to opnd2, and Z to the result. The value may be either S (single-precision) or D (double-precision). The three operand specification may be abbreviated according to the following rules:

- If no precision is specified, all operands are single precision.

- If a single letter (S or D) is specified, it applies to the first operand and result, with the second operand defaulted to single precision.

- If two letters are specified, 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 ADD 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	-

Note: Operand 2 is either one or two words depending on the precision specified with the keyword PREC. The total length of operand 1 is determined by the operand 1 precision multiplied by the value in the count operand.

Example

```
ADD  #1,2                   add 2 to index register 1

ADD  E,15,PREC=D            add 15 to double-prec value

ADD  V1,A,3,RESULT=V2       add the value in A to each
                            of 3 words starting at V1
                            and place the results in 3
                            words starting at V2.  V1
                            and A remain unchanged.
```

ADDV

Data Manipulation

The add vector instruction (ADDV) is used to add the components of operand 2 to the corresponding components of operand 1. Consecutive variables contained in operand 2 are added to the corresponding variables contained in operand 1.

Note: An overflow condition is not indicated by EDX.

Syntax

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

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

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

Operations On Index Registers

Index registers may generally be treated as ordinary single-precision integer arithmetic or logical variables. However, results of a vector operation directed at the registers, #1 and #2 may not extend beyond #2.

Example

```
V1    DATA     32F'1'
V2    DATA     32F'2'
      ADDV     V1,V2,32       add V2 to V1, 32 values
(After execution, V1 contains 32F'3')

      ADDV     #1,V3,2        add V3 to #1 and V4 to #2
V3    DATA     F'1'
V4    DATA     F'2'

(#1 is incremented by 1 and #2 is incremented by 2.)
```

AND

Data Manipulation

The AND instruction causes a logical anding together of the bit positions in operand 2 to operand 1. The operands are treated as bit strings and a comparison of each of the corresponding bits in each string is made. If the operand bits are both 1, the corresponding result bit is also set to 1. If either or both of the operand bits is a 0, the corresponding bit in the result is set to 0.

Syntax

```
label       AND     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. The length of opnd1 is determined by multiplying count times precision.
opnd2	The value by which the first operand is modified. Either the name of a variable or an explicit constant may be specified.
count	The number of consecutive variables in opnd1 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
```

RESULT= This optional operand represents 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.

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

Example

```
  AND     A,X'00FF'       AND bit positions of the constant
                          X'00FF with variable A

  AND     B,A,(1,BYTE)    AND bit positions of A with B
```

In the following example a mask value is ANDed with a data field to turn off the low order 4 bits in the data byte without affecting the other bits. After execution of the AND, the field DATA contains X'E0' (binary 1110 0000).

```
          AND    DATA1,MASK,(1,BYTE)
           .
           .
  DATA    DC     X'E7'    binary 1110 0111
  MASK    DC     X'F0'    binary 1111 0000
```

ATTACH

Task Control

The ATTACH instruction activates execution of another task. If the named task is already in the attached state, no operation occurs.

The task to be attached is normally assumed to be in the same partition as the ATTACH instruction. However, it is possible to ATTACH a task in another partition using the cross-partition capability of ATTACH. For more information refer to "Cross-Partition Services" in the System Guide.

When an ATTACH statement is issued, the address of either the default terminal or the currently active terminal for the task issuing the ATTACH, is placed into $TCBCCB of the target task. Therefore, the same terminal is active for both tasks.

Syntax

```
label       ATTACH    taskname,priority,CODE=value,
                      P1=,P2=,P3=

Required:   taskname
Defaults:   CODE=-1
Indexable:  none
```

Operands	Description
taskname	Name of the task to be attached. This task must be defined with a TASK statement.
priority	A priority to be assigned to the task. This priority will override and replace the one originally assigned in the TASK statement. It remains in effect unless superceded by a subsequent ATTACH statement. See the description of "TASK" on page 285 for a complete definition of priority.
CODE=	A code word to be inserted in the first word of the task control block of the task being attached. The code word may be tested in the attached task by referring to the taskname operand. Sometimes when

a task is attached from more than one point, it may be desirable to inform the task of the origin of the attachment. The code word value provides a simple mechanism for accomplishing this. Note that the code word should be examined immediately upon entry to the attached task, since execution of certain instructions (for example, I/O instructions) will cause the task code word to be overlaid.

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

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, this statement should not be placed between executable instructions. If the command entered is specified in the list, control will be passed to the associated user routine. This provides you with a mechanism for interactive control of programs from a terminal. These routines should be short because they are executed on hardware interrupt level 1; therefore, they may interfere with the execution of any other user programs. They must end with the ENDATTN instruction.

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 <u>Utilities, Operator Commands, Program Preparation, Messages and Codes</u> cannot also be defined in an ATTNLIST in the program to be tested.

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.

Example

```
         ATTNLIST    (PC1,PCODE1,PC2,PCODE2)

PCODE1   MOVE        CODE,1     ENTER HERE BY PRESSING
         ENDATTN                ATTENTION AND KEYING 'PC1'

PCODE2   POST        EVENT,2    ENTER HERE BY PRESSING
         ENDATTN                ATTENTION AND KEYING 'PC2'
```

Figure 4 shows the functional flow when ATTNLIST is used. Also see "Example 7: A Two Task Program With ATTNLIST" on page 395.

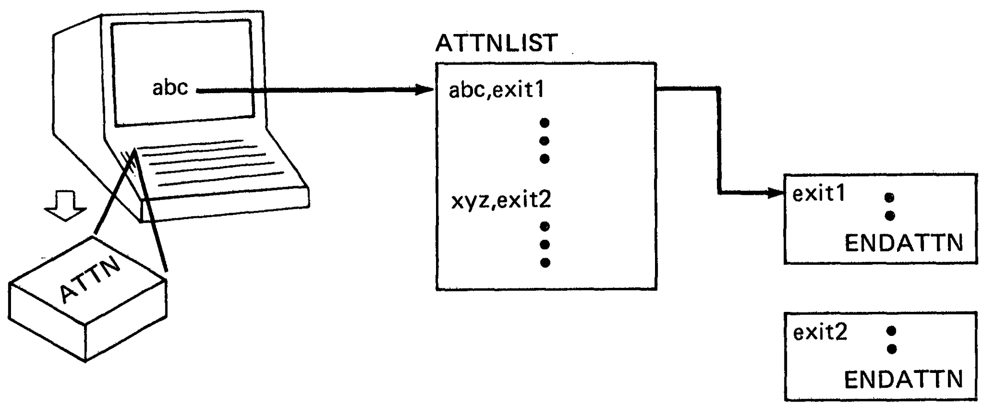


Figure 4. Function of ATTNLIST

BSC (BINARY SYNCHRONOUS COMMUNICATIONS)(REFERENCE ONLY)

BSCCLOSE

BSCREAD

BSCIOB

BSCOPEN

BSCWRITE

The Binary Synchronous instructions are described in detail in the Communications and Terminal Applications Guide

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.

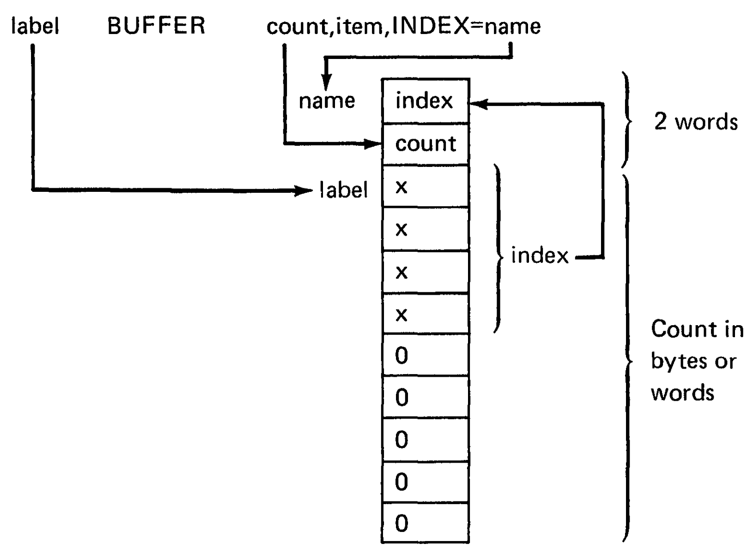
item — Buffer type indicator. Code BYTE or BYTES if the buffer length is defined in terms of bytes. Code WORD or WORDS if the buffer length is defined in terms of words. The default for this operand is WORD.

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.

Standard BUFFER



TPBSC BUFFER

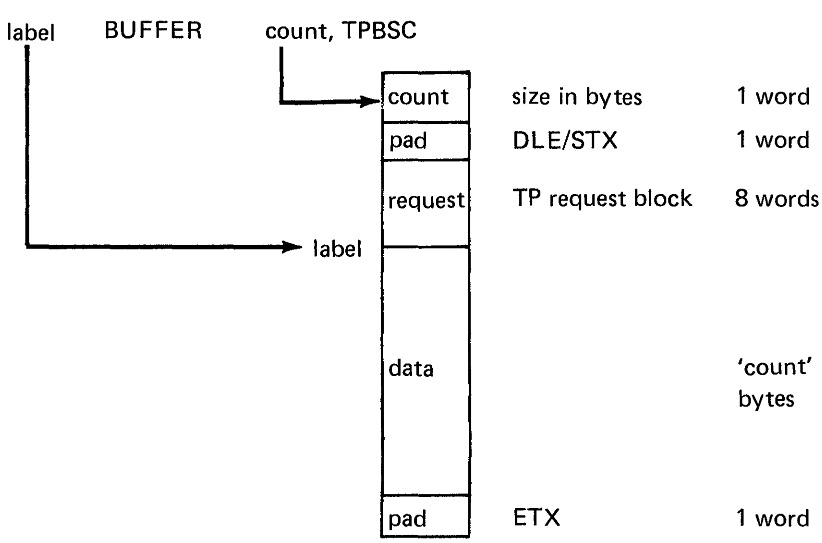


Figure 5. BUFFER Statement

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. Up to five, explicit, single precision, integer constants or the symbolic labels of single-precision integer variables which will be passed to the subroutine. 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 reference 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 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.

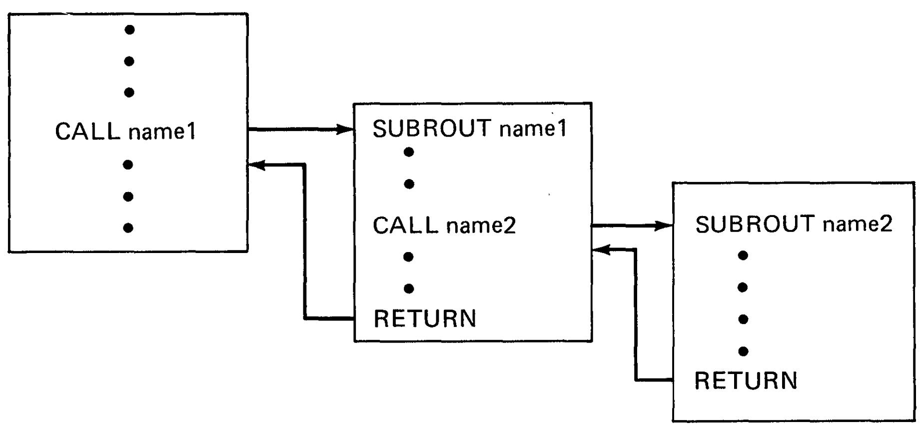


Figure 6. Execution of Subroutines

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.

Example

```
CALLFORT   FPGM1                  No parameters passed

CALLFORT   FSUB1,A                One parameter passed

CALLFORT   FSUB2,(A,B)            Two parameters passed

CALLFORT   FSUB2,(A,B),
           P=(INPUT,OUTPUT)       Two parameters
                                  passed with labels,
                                  INPUT for parameter A
                                  OUTPUT for parameter B
```

CONCAT

Graphics

The CONCAT statement concatenates two text strings, text1 and text2, or a text string and a graphic control character. Text from text2 is placed at the right of any text which is currently in the buffer text1 and the resulting text string is placed in text1. The character count of text1 is then changed to reflect the combined counts of the beginning contents of text1 plus the concatenated characters from text2. Truncation on the right occurs if the combined counts exceed the physical length of text1. You have the option to reset the character count of text1 to 0 before beginning to concatenate a new string.

Syntax

```
label       CONCAT   text1,text2,RESET,REPEAT=,P1=,P2=

Required:   text1,text2
Defaults:   REPEAT=1
Indexable:  none
```

Operands	Description
text1	Label of left input and resultant text.
text2	Label of right input text, an explicit 1-character constant (left-justified, for example C'A' or X'07'), or a symbol representing one of the following ASCII graphic control characters: GS, BEL, ESC, ETB, ENQ, FF, CR, LF, SUB, or US.
RESET	An indicator to reset the character count of text1 before starting the specified concatenation. No reset is done if this parameter is omitted.
REPEAT=	The number of times text2 is to be concatenated to text1. For example if a C' ' is coded as text2 and REPEAT is coded with a 5, then 5 blanks are concatenated to text1. REPEAT must be an absolute numeric value.

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

Note: See "Example 12: Graphics Instructions Programming Example" on page 408 for typical use of this instruction.

CONTROL

Tape Control

The CONTROL statement allows you to execute tape functions. You can space forward or backward a specified number of records or files (a file is the data between the beginning tapemark and the ending tapemark). You can also write tape marks, rewind the tape, set the tape drive offline, or rewind and set offline.

CONTROL also is used to close tape data sets. It is a recommended procedure to close all tape data sets. If you do not close data sets, then you must control the tape drive directly with the various CONTROL functions. Close to a SL (standard label) output tape will write the following trailer label: TM EOF1 TM TM. Close to a NL (no label) tape will write: TM TM. Input tapes are automatically rewound as the result of a close operation. An attempt to WTM (write tapemark) to an unexpired file (expiration date in the header label is not equal or less than the current date) is an error condition.

Syntax

```
label       CONTROL  DSx,type,count,END=,ERROR=,WAIT=,P3=

Required:   DSx,type
Defaults:   count=1,WAIT=YES
Indexable:  count
```

Operands	Description
DSx	x specifies the relative data set number in a list of data sets defined by you 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.
type	The type field is the CONTROL function to be performed. Following is a list of functions available:

FSF Forward space file (tapemark). Regardless of where the tape is currently positioned, the tape will search forward the number of tape marks indicated in the count operand. If the sepcified number of tape marks indicated by the count field are not on the tape, the positioning of the tape is unpredictable.

BSF Backward space file (tapemark). The tape will search backward until the next tape mark is read. The default value for count is 1. If the tape is at load point when when this command is issued, the load point return code is returned.

FSR Forward space record. The tape will space forward past the number of records specified in the count field. The default value for count is 1.

BSR Backward space record. The tape will space backward past the number of records specified in the count field. The default value for count is 1. If the tape is at load point when this command is issued, the load point return code is returned.

WTM Write tapemark. This function will write a tape mark on tape. If the count field is coded, successive tape marks will be written according to the count value.

REW Rewind tape to load point (beginning of tape).

ROFF Rewind tape and set the tape drive to offline.

OFF Set tape drive to offline.

CLSRU Close tape data set and allow it to be reused (reopened by another program or task without an intervening $VARYON command). The tape is repositioned to the HDR1 label of the data set for labeled tape. The tape is positioned to the beginning of the first data record for no label tapes. You can use $VARYON to change the file number being processed or you can use a CONTROL function.

CLSOFF Close tape data set, rewind tape, and set the tape drive to offline.

count The count operand specifies the number of files or records to be skipped or the number of tapemarks to be written. This can be a constant or the label of a count value.

END= Use this keyword 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.

If END is not coded, a tapemark will also be treated as an error. The physical position of the tape, under this condition, is the read/write head position is immediately following the tapemark. See CONTROL close functions for repositioning of the data set. Remember also that the count field might not be decremented to zero.

ERROR= Use this keyword to specify the first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If this operand is not specified, control will be returned to the next instruction after the READ and you must test the return code in the task code word for errors. This operand must not be used if WAIT=NO is coded.

WAIT= If this operand is allowed to default or if it is coded as WAIT=YES, the current task will be suspended until the operation is complete. If the function selected is CLSRU or CLSOFF then WAIT=YES is the only valid option for this operand, any other option will be ignored.

For functions other than close, if the operand is coded as WAIT=NO, control will be returned after the operation is initiated and a subsequent WAIT DSx must be issued in order to determine when the operation is complete.

END and ERROR cannot be coded if WAIT=NO is coded. You must subsequently test the return code in the Event Control Block (ECB) named DSx or in the task code word (referred to by 'taskname'). Two codes are of special significance. A -1 indicates a successful end of operation. A +10 indicates an 'End of Data Set' and may be of logical significance to the program rather than being an error. For programming purposes, any other return codes should be treated as errors.

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

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

```
CONTROL  DS1,CLSOFF
```

This statement closes the tape data set specified by DS1, rewinds the tape, and sets the tape drive offline.

```
CONTROL  DS2,FSR,16
```

This statement causes the tape data set specified by DS2 to be forward spaced 16 data records.

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 7 numeric digits
. = 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.

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

Data Formatting

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	7 decimal digits*
Extended-precision floating-point	16 decimal digits*

*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 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	No data in field
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 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.

If any other character is found during the conversion, the following action will be taken:

- For delimiters , or /

  End of field will be generated. If no data was found, a "Field Omitted" (2) will be returned.

- For all blanks

  "No Data in Field" (1) will be returned.

- For any other character (for example, an alphabetic character).

  "End of Field" (1) will be returned.

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.

CSECT

Program Module Sectioning

The CSECT statement names a program module to identify its location within the program output from $LINK.

The CSECT instruction is optional and if it is omitted the program module has a blank name.

Program modules, assembled by $EDXASM, can have multiple CSECT statements. However, all CSECTS, after the first one, will generate ENTRY instead of CSECT definitions.

Program modules assembled by means of the Series/1 Macro Assembler or host assembler are also permitted to have multiple CSECT instructions in a single assembly. These assemblers will generate a separate program module for each uniquely named CSECT.

Syntax

```
label       CSECT

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

Operands	Description
none	
label	The label must be the name of the program module for the first CSECT. For subsequent CSECTs the label must be an entry name.

DATA/DC

Data Definition

The DATA/DC statement defines one or more constants. Constants can have various forms of data representation such as binary, decimal, hexadecimal, character, floating-point, or address. Character strings or multiple constants may be defined in one DATA statement. The maximum number of bytes allowed in the value operand depends upon the program preparation facility used and can be determined by referencing the appropriate documentation. When using $EDXASM, up to 10 separate data specifications may be made on a DATA statement by separating the individual specifications with commas. When using $S1ASM, one data specification is allowed with each DATA statement.

Syntax

```
label        DATA      dup type value

label        DC        dup type value


Required:    type, value
Defaults:    dup=1
Indexable:   none
```

Operands	Description
dup	Duplication factor for the type constant defined.
type	Constant type or form of data representation.
value	The value to be assigned to the constant. Also determines field length of some types of constants. The value is enclosed in quotes for all constant types except A, in which the value is enclosed in parentheses.

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	7 decimal digits *
Extended-precision floating-point	16 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 7 numeric digits
. = 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 floating point in exponent form
MANY	DATA	F'1',D'2'	A word of 1 and a double word of 2

DCB

EXIO Control

The DCB statement creates a standard device control block (DCB) for use with EXIO. For additional information on DCBs refer to the description manual for the processor in use.

Syntax

```
label       DCB    PCI=,IOTYPE=,XD=,SE=,DEVMOD=,DVPARM1=,
                   DVPARM2=,DVPARM3=,DVPARM4=,CHAINAD=,
                   COUNT=,DATADDR=

Required:   label
Defaults:   PCI=NO,IOTYPE=OUTPUT,XD=NO,SE=NO
Indexable:  none
```

Operands	Description
PCI=	An interrupt indicator. Code PCI=YES to cause the device to present an interrupt at the completion of the DCB fetch prior to data transfer.
IOTYPE=	An indicator showing the type of operation. Code IOTYPE=INPUT for operations involving transfer of data from device to processor or for bidirectional transfers under one DCB operation. Code IOTYPE=OUTPUT for operations involving transfer of data from processor to device or for control operations involving no data transfer.
XD=	A DCB type indicator. Code XD=YES to indicate the DCB is a non-standard type.
SE=	An exception reporting indicator. Code SE=YES to indicate the device is allowed to suppress the reporting of certain exception conditions.
DEVMOD=	The byte that describes functions unique to a particular device. Code two hexadecimal digits.

DVPARM1= The value of device-dependent parameter word 1. Code as four hexadecimal digits or the label of an EQU preceded by a +.

DVPARM2= The value of device-dependent parameter word 2. Code as four hexadecimal digits or the label of an EQU preceded by a +.

DVPARM3= The value of device-dependent parameter word 3. Code as four hexadecimal digits or the label of an EQU preceded by a +.

DVPARM4= The value of device-dependent parameter word 4. Code as four hexadecimal digits or, if SE=YES, the label of the first byte to which residual status data is to be transferred. The length of the residual status area is device dependent.

CHAINAD= The label of the next DCB in the chain if chained DCBs are desired.

COUNT= The number of data bytes to be transferred. Code a decimal number between 0 and 32767 inclusive or the label of an EQU preceded by a +.

DATADDR= The label of the first byte of data.

For information on the contents of DVPARM1-DVPARM4 and DEVMOD refer to the description manual of the device to be used.

The example below shows two chained DCBs. WR1DCB is for some type of output operation in which the 120 byte field MSG1 will be transferred to the device. Any status information resulting from the operation will be placed in RESTAT by the device. WR2DCB is for some type of device control operation because it too defaulted to IOTYPE=OUTPUT but no data transfer (DATADDR=,COUNT=) was specified. RESTAT is used for status of this operation as well.

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,DEVMODE=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 of the QD will be assigned the name specified in the label on the DEFINEQ statement.
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.

DEQ

Task Control

The DEQ instruction releases exclusive control of a system or user resource other than a terminal. You must always dequeue any resource previously enqueued (ENQ). Failure to dequeue the resource prevents its further use. For additional information refer to the description of ENQ.

DEQ normally assumes that the QCB for the resource is defined in the same partition as the current program. However, it is possible to dequeue from a resource in another partition. For additional information. refer to the topic on "Cross-Partition Services" in the System Guide.

When using the $S1ASM macro assembler or the host assembler, the DEQ instruction causes the QCB defining the named resource to be generated at the end of the program. When using $EDXASM, no QCB will be generated; the QCB must be explicity created with the QCB instruction.

Syntax

```
label       DEQ     resource,code,P1=,P2=

Required:   resource
Defaults:   code=-1
Indexable:  resource
```

Operands	Description
resource	The symbolic name of the resource being dequeued. This must be the same name used for the ENQ instruction and is usually the label of a QCB statement.
code	A code word to be inserted into the queue control block (QCB) which defines the resource. The code word may be examined by referencing the symbolic name of the resource. This code may be used as a flag to indicate a status or a condition. A code of 0 is interpreted by the ENQ instruction to mean that the resource is unavailable for use; all non-zero

codes indicate the resource is available for other uses.

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

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 task code word of the task being detached. It is the first word of the task control block.
P1=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

DIVIDE

Data Manipulation

The DIVIDE instruction provides for signed division of opnd1 by opnd2. The remainder is stored in the task code word and will be lost after the next DIVIDE, I/O operation, or other operation that updates the task code word. Only if the divisor (opnd2) is double-precision will the remainder be double-precision. Divide overflow is indicated by the special remainder X'8000'. X'8000' is also the result of a divide by zero operation.

Note: An overflow condition is not indicated by EDX.

Syntax

```
label       DIVIDE    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. This is the dividend.
opnd2	The value by which the first operand is modified, either the name of a variable or an explicit constant. This is the divisor.
count	The number of consecutive variables upon which the operation is to be performed. The maximum value is 32767.
RESULT=	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. This operand is optional.

PREC=XYZ The precision value X applies to opnd1, Y to opnd2, and Z to the result. The value may be either S (single-precision) or D (double-precision). The Three operand specification may be abbreviated according to the following rules:

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

Example

```
DIVIDE VAL,(TAB,#1)    second operand indexed
```

DO

The DO instruction initializes a loop. A loop is a set of one or more instructions that are executed repetitively until the condition specified by the DO is satisfied. The DO loop must have an associated ENDDO instruction which defines the end of the loop. There are three forms of the DO instruction. DO UNTIL and DO WHILE provide a means of looping until or while a relational statement is true. The third form of the DO instruction causes a loop to be executed a specific number of times. In all of these forms a branch out of the loop is allowed.

Note: Because coding practice is to code DO and ENDDO together, the description of ENDDO is duplicated immediately following the DO description for convenience.

Examples of DO and ENDDO are shown at the end of this section.

Syntax

```
label       DO    count,TIMES,INDEX=,P1=

label       DO    UNTIL,statement

label       DO    WHILE,statement

Required:   count or one relational statement
            with UNTIL or WHILE
Defaults:   none
Indexable:  count or data1 and data2 in each statement
```

Operands	Description
count	The number of times the loop is to be executed. It is an explicit constant, or the label of a count. The maximum value is 32767. Note: If count=0, then the loop will be executed one time.

TIMES — An optional operand which only serves to comment the instruction for program readability.

INDEX= — The label of a variable, defined by the user, which will be reset to 0 before starting the DO loop and will be incremented by 1 immediately prior to each execution of the instruction following the DO instruction. Therefore, the first time the loop is executed the index will have a value of 1.

UNTIL — This parameter establishes a trailing decision loop, which is executed until the exit condition is true. Even if the condition is true initially, the loop will be executed one time.

WHILE — This parameter establishes a leading-decision loop, which is executed as long as the exit condition is true. Note that if the condition is false initially, the loop will not be executed.

statement — A relational statement or statement string indicating the condition for the loop exit. This form is valid only following UNTIL or WHILE.

Note: Additional details such as coding the operands data1 and data2 in a relational statement are described following "Program Sequencing Instructions" on page 34. For examples of relational statements see "Examples of Relational Statements" following the descriptions of "IF" on page 177.

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

ENDDO

Program Sequencing

The ENDDO instruction defines the end of a DO loop. It must be preceded by a DO instruction. Up to twenty nested loops are allowed, and each must be defined by a DO and an ENDDO.

Syntax

```
label        ENDDO

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

Operands	Description
none	none

Example of DO and ENDDO

1. Simple DO

```
      DO    100
         :
         (execute 100 times)
      ENDDO
```

2. Simple DO with TIMES coded

```
      DO    N,TIMES
        :
        (execute 'N' times)
      ENDDO
```

3. DO UNTIL

```
     DO      UNTIL,(A,EQ,1000,FLOAT)
       :
       (execute until A EQ 1000)
     ENDDO
```

4. DO WHILE

```
     DO      WHILE,(B,NE,C)
       :
       (execute while B NE C)
     ENDDO
```

5. Nested DO loops

```
     DO      UNTIL,(A,EQ,B,DFLOAT),OR,(#1,EQ,1000)
       :
       DO     10,TIMES
         :
       ENDDO
     ENDDO
```

6. Nested DO loops and IF statements

```
     DO      WHILE,(A,GT,B,DWORD)
       IF      (CHAR,EQ,C'A',BYTE)
          DO      40,TIMES
            :
          ENDDO
       ELSE
         :
       ENDIF
     ENDDO
```

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 control block. Fields within these DSCB 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 PROGRAM.

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

Syntax

```
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 will not be needed for writing 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.

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.

EJECT

Listing Control

The EJECT statement causes the next line of the listing to appear at the top of a new page. This statement provides a convenient way to separate sections of a program. It does not change the page title if one is in force.

Syntax

```
blank     EJECT
```

Operands	Description
none	none

ELSE

Program Sequencing

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

Note: Since IF, ELSE, and ENDIF are usually coded together, this description is repeated for your convenience following the IF instruction.

Syntax

```
label        ELSE

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

Operands	Description
none	none

Example: The examples for IF, ELSE, and ENDIF are shown following the IF instruction.

END

Task Control

The END statement must be the last statement coded in your program.

Syntax

```
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.

ENDDO

Program Sequencing

The ENDDO instruction defines the end of a DO loop. It must be preceded by a DO instruction. Twenty nested loops are allowed, and each must be defined by a DO and an ENDDO. Examples of DO loops are shown following the description of "DO" on page 101.

Note: Because the practice is to code DO and ENDDO together, this instruction is repeated following the DO instruction.

Syntax

```
label       ENDDO

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

Operands	Description
none	none

Example: See the examples following the DO instruction.

ENDIF

Program Sequencing

The ENDIF instruction 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.

Note: Since IF, ELSE, and ENDIF are usually coded together, this description is repeated for your convenience following the IF instruction.

Syntax

```
label       ENDIF

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

Operands	Description
none	none

Example: Examples of IF, ELSE, and ENDIF are shown following the IF instruction.

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 task code word (first word of the TCB) of the task being detached.
P1=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

ENQ

Task Control

The ENQ instruction acquires exclusive control of a system or user resource other than a terminal.

A resource is a logical or physical entity (for example an I/O device, subroutine, or data set) which must be used in a serial fashion. Enqueuing is the process of acquiring exclusive control in order to ensure serial (one at a time) use. In general, there are two types of resources, system and user. System resources are those which may be shared serially by all user programs, and are defined by symbolic names which are known broadly across the system. User resources are shared serially by different parts of one user program and are identified by symbolic names known only within that user program.

Syntax

```
label       ENQ     resource,BUSY=busyaddr,P1=

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

Operands	Description
resource	The symbolic name of the resource to be enqueued.
BUSY=	The address of the instruction to receive control if the requested resource is not available. If the resource is busy and this operand is not specified, the requesting task will be placed in a wait state until it is available.
P1=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Each named resource is represented by a 5-word QCB. The resource name is the label of the QCB. You must explicitly code any QCBs necessary in programs to be assembled with $EDXASM. The Series/1 and host macro assemblers automatically create the necessary QCB if a DEQ instruction naming the resource is included in the program.

ENQ normally assumes that the resource (QCB) to be queued for is in the same partition as the current program. However, it is possible to enqueue on a resource in another partition using the cross-partition capability of ENQ. For more information on this subject refer to the System Guide topic on "Cross-Partition Services."

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.
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 serv-

iced. 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.
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

RESULT= 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. This operand is optional.

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

Example

```
C     DATA    X'92'                   binary 10010010
D     DATA    X'8F'                   binary 10001111
R     DATA    X'00'
      EOR     C,D,(1,BYTE),RESULT=R
```

After execution of the example EOR, fields C and D are unchanged. Field R looks like this:

```
R    DATA   X'1D'        binary 00011101
```

EQU

Data Definition

The EQU instruction assigns a value to a symbol. The symbol (the label on the EQU statement) can be used as an operand in other instructions wherever symbols are allowed.

Syntax

```
label         EQU       value

Required:     label,value
Defaults:     none
Indexable:    none
```

Operands	Description
value	A self-defining term or another symbol. If it is a symbol it must have been previously defined. The symbol may be coded as an asterisk (*). The asterisk refers to the next available storage location in the program. It is used primarily to generate convenient labels for use within the program.

Note: When the symbol is used as an operand in an instruction that allows either immediate data or the label of a variable as the operand, the symbol will be interpreted as a variable unless it is preceded by a plus (+) sign.

The label may be used in other instructions as desired. When using $EDXASM it must be preceded by a + where literal or immediate data is desired; otherwise, it is assumed to be the address of the data.

Example

```
A       EQU 2              A has the value of 2
        MOVE (A,#1),7      7 is moved to addr (2 + #1)
        MOVE C,A           Contents of addr 2 moved to C
        MOVE C,+A          A '2' is moved to C

B       EQU  A             B also has the value of A (2)
        MOVE C,+B          A '2' is moved to C
CALLA   EQU  *             CALLA is equivalent to CALLSUB
CALLSUB CALL PROGA
```

ERASE

Terminal I/O

The ERASE instruction causes designated portions of the screen to be cleared (blanked) and set to a no data, null characters condition. It applies only to terminals accessed in STATIC mode. STATIC mode is specified with the SCREEN parameter of either a TERMINAL or IOCB statement.

Syntax

```
label       ERASE  count,MODE=,TYPE=,SKIP=,LINE=,SPACES=

Required:   none
Defaults:   count=maximum,MODE=FIELD,TYPE=DATA,
            SKIP=0,LINE=current line,SPACES=0
Indexable:  count,SKIP,LINE,SPACES
```

Operands	Description
count	The number of bytes to be erased. Both non-protected and protected characters contribute to the count, even if only non-protected characters are erased.
MODE=	The terminating condition for the erase operation. MODE=FIELD: The operation terminates whenever the mode-of-character display changes from non-protected to protected, or when the end of the current line is reached. MODE=LINE: Erasure continues to the end of the line. MODE=SCREEN: Erasure continues to the end of the logical screen. Exhaustion of the count takes precedence over any other terminating condition. An unspecified count is therefore implicitly large enough to include the entire logical screen.

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. 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.

EXOPEN

EXIO Control

EXOPEN is used to specify the locations where information is to be returned after an EXIO device interrupt. EXOPEN does not reset device status or device busy.

Syntax

```
label       EXOPEN    devaddr,listaddr,ERROR=,P1=,P2=

Required:   devaddr,listaddr
Defaults:   none
Indexable:  listaddr
```

Operands	Description
devaddr	The device address as two hexadecimal digits.
listaddr	The label of the first word of a list of three addresses.

The three addresses in the list are:

1. The address of a 3-word block where, after an interrupt, the system will store:

    a. Interrupt ID word

    b. Level status register at time of interrupt

    c. Address of ECB posted

    Note: If this word is zero, the information will not be returned.

2. The address of a list of ECB addresses. The interrupt condition code (ccode) received from the device will determine which ECB in the list will be posted. A ccode=0 will cause posting at the first ECB in the list, etc. The same ECB may be specified for more than one condition code. The ECB specified for ccode=2 (ex-

ception) will be posted in the event of a program error. The posting code contains:

a. Bit 0 on (1)

b. Bits 4-7 ccode

c. Bits 8-15 device address

Interrupt condition codes are shown in Figure 8 on page 132

3. The address of a DCB containing the parameters of a start cycle steal status operation. This operation will be started by the system, using this DCB, if an exception interrupt is received from this device. If the word is zero, the operation will not be performed.

ERROR= The label of the first instruction to be executed if an error is encountered during the execution of this instruction.

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

Note: Refer to the description manual for the processor in use for more information on interrupt ID, level status register, interrupt condition codes, and DCBs. Refer to the description manual for the device in use for information on the causes of various condition codes and the status information available using start cycle steal status.

EXIO Return Codes

I/O Instruction Return Codes are located in word 0 of TCB. Word 1 of TCB contains supervisor instruction. address.

Code	Description
-1	Command accepted
1	Device not attached
2	Busy
3	Busy after reset
4	Command reject
5	Intervention required
6	Interface data check
7	Controller busy
8	Channel command not allowed
9	No DDB found
10	Too many DCBs chained
11	No address specified for residual status
12	EXIODEV specified zero bytes for residual status
13	Broken DCB chain (program error)
16	Device already opened

Figure 7. EXIO Return Codes

Code	Description
0	Controller end
1	Program Controlled Interrupt (PCI)
2	Exception
3	Device end
4	Attention
5	Attention and PCI
6	Attention and exception
7	Attention and device end
8	Not used
9	Not used
10	SE on and too many DCBs chained
11	SE on and no address specified for residual status
12	SE on and EXIODEV specified no bytes for residual status
13	Broken DCB chain
14	ECB to be posted not reset
15	Error in Start Cycle Steal Status (after exception)

Note: Interrupt Condition Codes (Bits 4-7 of word 0 of ECB) (If bit 0 is on, bits 8-15=device ID)

Figure 8. EXIO Interrupt Codes

Example

```
L4OP     EXOPEN     E4,LNLIST
            .
            .
            .
LNLIST   DATA       A(LNID)
         DATA       A(LNECBS)
         DATA       A(LNSCSS)
            .
            .
LNID     DATA       3F'0'
LNECBS   DATA       F'0' no ECB for code 0
         DATA       A(LNPCIR)
         DATA       A(LNEXCP)
         DATA       A(LNDEVD)
            .
            .
            .
LNSCSS   DCB        IOTYPE=INPUT,COUNT=20,DATADDR=LNCSD
LNCSD    DATA       10F'0'
LNPCIR   ECB        0
LNEXCP   ECB        0
LNDEVD   ECB        0
```

EXTRN/WXTRN

Program Module Sectioning

Both of these statements identify symbols which are not defined within the program module containing the EXTRN/WXTRN statement. References to these symbols will be resolved when the program module is link-edited with a program module containing an ENTRY definition for the subject symbol. If no symbol is found during link-edit, the symbol is said to be unresolved and it is assigned the same address as the beginning of the program.

WXTRN symbols are resolved only by symbols that are contained in modules that are included by the INCLUDE statement in the link-edit process or by symbols found in modules called by the AUTOCALL function. However, WXTRN itself does not trigger AUTOCALL processing.

Only symbols defined by EXTRN statements will be used as search arguments during the AUTOCALL processing function of $LINK. Any additional external symbols found in the module found by AUTOCALL will be used to resolve both EXTRN and WXTRN symbols. See the description of $LINK in Utilities, Operator Commands, Program Preparation, Messages and Codes for further information.

Syntax

```
    blank      EXTRN   One or more relocatable symbols
    blank      WXTRN   that are external to this
                       program, separated by commas

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

Operands	Description
	One or more external symbols which will be resolved by link-editing to a program module which contains the same symbol defined by an ENTRY statement.

FADD

Data Manipulation

The floating-point ADD provides signed addition of operand 2 to operand 1. FLOAT=YES must be coded on the PROGRAM statement of any program whose initial task uses floating-point instructions and on the TASK statement of any task containing floating-point instructions.

Syntax

```
label       FADD     opnd1,opnd2,RESULT=,PREC=,
                     P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   RESULT=opnd1,PREC=FFF
Indexable:  opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies. For example, the variables in FADD A,B correspond to the common algebraic notation A+B. If the RESULT operand is not specified, then opnd1 is also the implicit result. This operand may not 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 integer constant (immediate data) between -32768 and +32767 may be specified.
RESULT=	This operand is optional and can be coded with the name of a variable in which the result is to be placed. When this operand is coded the variable specified by the first operand is not modified.
PREC=	All possible combinations of single and extended precision are permitted. An immediate value for opnd2 will be converted to a single-precision value regardless of any other method of precision specification discussed in the following paragraphs.

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

The PREC operand is specified as xyz where x, y, and z are characters representing the precision of opnd1, opnd2, and the RESULT operands respectively. Either 2 or 3 characters must be specified depending on whether or not the RESULT operand was coded. Permissible characters are:

F = Single-precision (32 bits)
L = Extended-precision (64 bits)
* = Default (single-precision)

If the precision of an operand is not established by the PREC operand, it will default to single-precision.

Return Codes: Floating-point operations produce return codes which are placed in the task code word, referred to by taskname (see PROGRAM/TASK). These codes must be tested immediately after the floating-point instruction is executed or the code may be destroyed by subsequent instructions.

Code	Description
-1	Successful completion
1	Floating point overflow
5	Floating point underflow

Examples:

```
FADD    F1,F2,RESULT=F3
FADD    (0,#1),(2,#2),RESULT=ANSL,PREC=LLL
FADD    VALUE,32767,PREC=LF
```

FDIVD

Data Manipulation

Floating-point divide provides signed division of operand 1 by operand 2. FLOAT=YES must be coded on the PROGRAM statement of any program whose initial task uses floating-point instructions and on the TASK statement of any task containing floating-point instructions.

Syntax

```
label       FDIVD     opnd1,opnd2,RESULT=,PREC=,
                      P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   RESULT=opnd1,PREC=FFF
Indexable:  opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies. If the RESULT operand is not specified, then opnd1 is the implicit result. This operand must not 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 integer constant (immediate data) between -32768 and +32767 may be specified.
RESULT=	This operand is optional and can be coded with the name of a variable in which the result is to be placed. In this case, the variable specified by the first operand is not modified.
PREC=	All possible combinations of single and extended precision are permitted. An immediate value for opnd2 will be converted to a single precision value regardless of any other method of precision specification discussed in the following paragraphs.

The PREC operand is specified as xyz where x, y, and z are characters representing the precision of opnd1, opnd2, and the RESULT operands respectively. Either 2 or 3 characters must be specified depending on whether or not the RESULT operand was coded. Permissible characters are:

F = Single-precision (32 bits)
L = Extended-precision (64 bits)
* = Default (single-prcision)

If the precision of an operand is not established by the PREC operand, it will default to single-precision.

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

Return Codes: Floating-point operations produce return codes which are stored in the task code word, referred to by taskname (see PROGRAM/TASK). The codes must be tested immediately after the floating-point instruction is executed or the code may be destroyed by subsequent instructions.

Code	Description
-1	Successful completion
1	Floating point overflow
3	Floating point divide check (divide by '0')
5	Floating point underflow

Examples:

```
FDIVD   DIV1,DIV2,RESULT=ANS
FDIVD   AB,300,PREC=LS
```

FIND

Program Sequencing

FIND is used to locate the first occurrence of a specific character (byte) in a character (byte) string.

Syntax

```
label        FIND     character,string,length,where,
                      notfound,DIR=,P1=,P2=,P3=,P4=,P5=

Required:    character, string, length, where, notfound
Defaults:    DIR=FORWARD
Indexable:   string, length, and where
```

Operands	Description
character	Specify the character that is the object (target) of the search. If searching for an EBCDIC alphameric character, specify it in the format C'x' where x is the desired character. For a bit string which is not an alphameric character, specify as X'xx'.
string	Specify the address of the string to be searched.
length	Specify the length of the string to be searched. Either the name of a variable or an explicit integer constant (immediate data) may be specified.
where	Specify the location where the address of the target character is to be stored if it is found. If it is not found, this word will be unchanged.
notfound	Specify the address of the instruction to be executed if the target character is not found.
DIR=	Specify DIR=FORWARD or omit to search from left to right. Specify DIR=REVERSE to search from right to left.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example

```
FIND      C'$',MSG1,20,POINTER,NOTFOUND

FIND      X'A0',(0,#1),LSTR,POINTER,NOGOOD
```

FINDNOT

Program Sequencing

FINDNOT is used to find, in a character string, the first occurrence of a character (byte) different from the one specified.

Syntax

```
label       FINDNOT    character,string,length,where,
                       notfound,DIR=,P1=,P2=,P3=,P4=,P5=

Required:   character, string, length, where, notfound
Defaults:   DIR=FORWARD
Indexable:  string, length, and where
```

Operands	Description
character	Specify the character you are searching for. If searching for an alphameric character specify it in the format C'x' where x is the desired character. For a bit string which is not an alphameric character, specify as X'xx'.
string	Specify the address of the string to be searched.
length	Specify the length of the string to be searched. Either the name of a variable or an explicit integer constant (immediate data) may be specified.
where	Specify the location where the address of the first non-target character is to be stored if it is found. If one is not found, this word will be unchanged.
notfound	Specify the address of the instruction to be executed if a non-target character is not found.
DIR=	Specify DIR=FORWARD or omit to search from left to right. Specify DIR=REVERSE to search from right to left.

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

Example

```
FINDNOT  C' ',INPUT,80,CPOINTER,ALLBLANK

FINDNOT  X'40',CARD+79,80,LASTCHAR,ALLBLANK,DIR=REVERSE
```

FIRSTQ

Queue Processing

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

Syntax

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

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

Operands	Description
qname	The name of the queue from which the entry is to be fetched. The queue name is the label of the DEFINEQ instruction which created the queue.
loc	The address of one word of storage where the entry is placed. #1 or #2 can be used.
EMPTY=	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 FIRSTQ 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.

Example: See the example of queuing instructions in the example following the NEXTQ instruction.

FMULT

Data Manipulation

This instruction provides signed floating-point multiplication of operand 1 by operand 2. FLOAT=YES must be coded on the PROGRAM statement for programs whose initial task uses floating-point instructions and on the TASK statement of every task containing floating-point instructions.

Syntax

```
label        FMULT     opnd1,opnd2,RESULT=,PREC=,
                       P1=,P2=,P3=

Required:    opnd1,opnd2
Defaults:    RESULT=opnd1,PREC=FFF
Indexable:   opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies. If the RESULT operand is not specified, then opnd1 is also the implicit result. This operand may not 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 integer constant immediate data between -32768 and +32767 may be specified.
RESULT=	This operand may optionally be coded with the name of a variable in which the result is to be placed. In this case, the variable specified by the first operand is not modified.
PREC=	All possible combinations of single and extended precision are permitted. An immediate value for opnd2 will be converted to a single precision value regardless of any other method of precision specification discussed below.

The PREC operand is specified as xyz; where x, y, and z are characters representing the precision of opnd1, opnd2, and the RESULT operands respectively. Either 2 or 3 characters must be specified depending on whether or not the RESULT operand was coded. Permissible characters are:

F = Single-precision (32 bits)
L = Extended-precision (64 bits)
* = Default (single-precision)

If the precision of an operand is not established by the PREC operand, it will default to single-precision.

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

Return Codes: Floating-point operations produce return codes in the task code word, referred to by taskname (see PROGRAM/TASK). These codes must be tested immediately after the floating-point instruction is executed or the code may be destroyed by subsequent instructions.

Code	Description
-1	Successful completion
1	Floating point overflow
5	Floating point underflow

Example

```
FMULT    F1,F2
FMULT    A,B,PREC=FLL,RESULT=DOUBLE
```

FORMAT

Data Formatting

Specifies the type of conversion to be performed when data is transferred from storage to a text buffer by a PUTEDIT instruction, or from a text buffer to storage by a GETEDIT instruction.

The FORMAT statement must be contained in the assembly in which it is referenced and cannot be placed within a sequence of executable program instructions.

Note: The FORMAT statement may be continued on multiple lines but each line (except the last) must be coded through column 71 and must have a continuation symbol in column 72. Commas may not be used to continue a line before column 71.

Syntax

```
label       FORMAT   list,gen

Required:   list
Defaults:   gen=BOTH
Indexable:  none
```

Operands	Description
list	Conversion specifications for the data to be converted. May be:

Item Type	Definition
I	Integer numeric
F	Floating point numeric
E	Floating point numeric E notation

H   Literal alphameric data, enclosed in quotes

X   Blanks

A   Alphameric data

gen   GET, if this FORMAT statement is for the exclusive use of GETEDIT instructions.

PUT, if this format statement is for the exclusive use of PUTEDIT instructions.

BOTH, if this format statement can be used with GETEDIT and PUTEDIT instructions. BOTH, the default, requires more storage than either GET or PUT.

The PUTEDIT statement retrieves each variable in the list, converts it according to the respective item specification in the format statement, and loads it into the text buffer specified. Spaces (blanks), line control characters, and literals may be inserted.

The GETEDIT statement moves data from the text buffer, converts it as specified in the FORMAT statement, and stores it at specified addresses. Characters in the input buffer may be skipped.

The slash (/) in a FORMAT statement associated with a GETEDIT statement acts as a delimiter, performing the same function as a comma.

Successive items in the buffer transfer list are converted and moved according to successive specifications in the FORMAT statement until all items in the list are transferred. If there are more items in the list than there are specifications in the FORMAT statement, control transfers to the beginning of the FORMAT statement and the same specifications are used again until the list is exhausted. The entire transfer is treated as a single record.

No check is made to see that the specifications in a FORMAT statement correspond in mode with the list items in the GETEDIT or PUTEDIT instructions. It is your responsibility to ensure that integer variables are associated with I-type format specification and real variables with F-type or E-type format specifications. You must also ensure that ample storage is available for transfer of data in a PUTEDIT operation.

Conversion of Numeric Data

The following specifications, or conversion codes, are available for the conversion of numeric data:

Item Type	Form	Definition
I	Iw	Integer numeric
F	Fw.d	Floating point numeric
E	Ew.d	Floating point numeric E notation

where:

w is an unsigned integer constant specifying the total field length of the data. This specification may be greater than that required for the actual digits in order to provide spacing between numbers; however, the maximum width allowed is 40 for I or F specifications.

d is an unsigned integer constant specifying the number of decimal places to the right of the decimal point. The allowable range is 0 to w-1 for F-type specifications and 0 to w-6 for E-type specifications.

Note: The decimal point between the w and d portions of the specification is required.

The following discussion of conversion codes deals with loading a text buffer, using PUTEDIT, in preparation for printing a line. The concepts, however, apply to all permissible text buffer operations.

Integer Numeric Conversion

General form: Iw

The specification Iw loads a text buffer with an EBCDIC character string representing a number in integer form; w print positions are reserved for the number. The number is right-justified. If the number to be loaded is greater than w-1 positions and the number is negative, an error condition will occur. A print position must be reserved for the sign if negative values are possible; however, positive values do not 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 fractional portion is truncated from the right. 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 the input character string.

If insufficient positions are reserved by d, the digits to the right of d digits are truncated. If excessive positions are reserved by d, zeros are filled in from the right for the insignificant digits.

The following examples show how each of the values on the left is converted according to the specification E10.4:

Internal Value	Value in the Buffer
12.17	b.1217Eb02
-41.16	-.4116Eb02
-.2	-.2000Eb00
7.3542	b.7354Eb01
-1.	-.1000Eb01
9.03	b.9030Eb01
.00187	b.1870E-02

Notes:

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

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

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

4. Numbers for E-conversion need not have their decimal points appearing in the input fields (in the text buffer). If no decimal point appears, you need not allocate space 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 E7.2, the following conversions would be performed:

Text Buffer Characters	Converted Internal Value
12.17E0	12.17
b1217E1	121.7
121.7E-2	1.217

Alphameric Data Specification

The following specifications are available for alphameric data:

Item Type	Form	Definition
H	'data'	Literal alphameric data
A	A	Alphameric data
X	X	Insert blanks (output) or skip input fields

The H-specification is used for alphameric data that is not changed by the program, such as printed headings.

The A-specification is used for alphameric data in storage which is to be operated on by the program such as a line that is to be printed.

The X-specification is used to bypass one or more input characters or to insert blanks (spaces) on an output line.

Literal Specification

General form: H

The H-specification is used to create alphameric constants. The maximum length for a literal is 255.

Literals must be enclosed in apostrophes. For example:

```
FORMAT ('INVENTORY REPORT')
```

The apostrophe (') and ampersand (&) characters within literal data are represented by two successive characters. For example, the characters DO & DON'T must be represented as:

```
DO && DON''T
```

Literal data can be used only in loading a text buffer; it is invalid in a GETEDIT statement. All characters between the apostrophes (including blanks) are loaded into the buffer in the same relative position they appear in the FORMAT statement. Thus:

```
FM   FORMAT ('THIS IS ALPHAMERIC DATA',3X,A6)
          .
          .
     PUTEDIT  FM,TEXT,(ALP)
```

cause the following record to be loaded into the buffer labeled TEXT.

```
THIS IS ALPHAMERIC DATA   AAAAAA
```

Literal data may also be included with variable data.

For example, the instructions:

```
FM   FORMAT  ('TOTAL OF',I2,' VALUES = ',F5.2)
          .
          .
     PUTEDIT  FM,TEXT,(TOTAL,VALUE)
```

cause a record such as the one in the following example to be loaded into the buffer.

```
TOTAL OF 5 VALUES = 35.42
```

Alphameric Specification

General form: Aw

The specification Aw is used to transmit alphameric data to/from variables in storage. It causes the first w characters to be stored into or loaded from the area of storage specified in the text buffer transfer list. For example, the statements:

```
FM   FORMAT (A4)
          .
          .
     GETEDIT  FM,TEXT,(ERROR)
```

cause four alphameric characters to be transferred from the buffer TEXT into the variable named ERROR.

The following statements:

```
FM    FORMAT  ('XY=',F9.3,A4)
          .
          .
      PUTEDIT  FM,TEXT,(A,ERROR,B,ERROR)
```

may produce the following line:

```
XY= 5976.000....XY= 6173.500....
```

In this example, the ellipses (....) represent the contents of the character string field ERROR.

The A-specification provides for storing alphameric data into a field in storage, manipulating the data (if required), and loading it back to a text buffer.

The alphameric field can be defined using the DATA statement or the TEXT statement. On input (GETEDIT) the alphameric field is set to blanks prior to data conversion. The alphameric data is left justified in the field.

Blank Specification

General form: X

The X-specification allow you to insert blank characters in an output buffer record and to skip characters of an input buffer record.

When the nX specification is used with an input record, n characters are skipped before the transfer of data begins. When the nX specification is used with an output record, n characters are inserted before the transfer of data begins. For example, if a buffer has four 10-position fields of integers, the statement:

```
FORMAT  (I10,10X,I10,I10)
```

could be used to avoid transferring the second field.

When the X-specification is used with an output record, n positions are set to blanks, allowing for spaces on a printed line. For example, the statement:

```
FORMAT  (F6.2,5X,F6.2,5X,F6.2,5X)
```

may be used to set up a line for printing as follows:

```
-23.45bbbbbb17.32bbbbbb24.67bbbbb
```

where b represents a blank.

Blank Lines in Output Records

Blank lines may be introduced between output records by using consecutive slashes (/). The slash causes a line control character to be inserted in the buffer. The number of blank lines inserted between output records depends upon the number and placement of the slashes within the statement.

If there are n consecutive slashes at the beginning or end of a format specification, n blank lines are inserted between output records. For n consecutive slashes elsewhere in the format specification, the number of blank lines inserted is n-1. For example, the statements:

```
       PUTEDIT  FM,TEXT,(X,(Y,D),Z)
         .
         .
  FM   FORMAT   ('SAMPLE OUTPUT',/,I5////I9,I4//)

  X    DC       F'-1234'
  Y    DC       D'111222333'
  Z    DC       F'22'
  TEXT TEXT     LENGTH=50
```

result in the following output:

```
  SAMPLE OUTPUT
  -1234
  (3 blank lines)

  111222333  22

  (2 blank lines)
```

Repetitive Specification

A specification may be repeated as many times as desired, within the limits of the text buffer size, by preceding the specification with an unsigned integer constant. The allowable range is 1 (the default) to 255.

Thus,

```
  (2F10.4)
```

is equivalent to:

```
  (F10.4,F10.4)
```

and uses less storage.

A parenthetical expression with multiplier (repeat constant) is permitted to enable repetition of data fields according to format specifications contained within the parentheses. All item types are permitted within the parenthetical expression except another parenthetical expression. Multiple parenthetical expressions may be specified within the same FORMAT statement. For example, the statement:

```
FORMAT   (2(F10.6,F5.2),I4,3(I5))
```

is equivalent to:

```
FORMAT   (F10.6,F5.2,F10.6,F5.2,I4,I5,I5,I5)
```

Storage Considerations

In general, the fewer items in the FORMAT list, the less storage that is required. An item is defined as a single conversion specification, literal data string, one or more grouped record delimiters, or a parenthetical multiplier. For example, the following format statements all have three items:

```
FORMAT  (I5,I5,I6)

FORMAT  (I5,3I5,'ITEM 3')

FORMAT  (3(I5),3I5)

FORMAT  (I5/,I5)

FORMAT  (I5,///,I5)

FORMAT  (/,/,/)

FORMAT  (2(/),/)

FORMAT  (2(1X),2X)

FORMAT  (I5/,2X)
```

FPCONV

Data Manipulation

FPCONV is used to convert integer values to or from floating-point numbers, by using the optional floating-point hardware feature. FLOAT=YES must be coded on the PROGRAM statement for programs whose initial task uses floating-point instructions and on the TASK statement of every task containing floating-point instructions.

Syntax

```
label        FPCONV  opnd1,opnd2,COUNT=,PREC=,
                     P1=,P2=,P3=

Required:    opnd1,opnd2
Defaults:    COUNT=1,PREC=FS
Indexable:   opnd1,opnd2
```

Operands	Description
opnd1	The address (label or index register reference) to receive the output of the conversion.
opnd2	The address of the data input to the conversion. 'opnd2' may also be immediate data in the form of an integer constant between -32768 and +32767.
COUNT=	The number of values, beginning at opnd2, to be converted and stored at locations beginning at opnd1. If opnd2 is immediate data, it will be converted and stored beginning at the location defined by opnd1 for as many locations as are defined by the COUNT operand.
PREC=	Defines the type and precision of opnd1 and opnd2 respectively. Its form is PREC=xy. The xy is a two character value composed of two of the following symbols. The type, integer or floating-point, of opnd1 and opnd2 must not be the same.

S = One word integer (or immediate data)
D = 2-word integer
F = Single-precision floating-point
L = Extended-precision floating-point
* = Use default (single-precision)

If PREC is not coded, the default specification for the operand will be used.

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

Example

```
FPCONV  A,B,COUNT=5,PREC=LD

FPCONV  X,L4,PREC=DL

FPCONV  (6,#1),C

FPCONV  (X,#1),(Y,#2),PREC=DL
```

FSUB

Data Manipulation

This instruction provides floating-point signed subtraction of operand 2 from operand 1. FLOAT=YES must be coded on the PROGRAM statement for programs whose initial task uses floating-point instructions and on the TASK statement of every task containing floating-point instructions.

Syntax

```
label       FSUB     opnd1,opnd2,RESULT=,PREC=,
                     P1=,P2=,P3=

Required:   opnd1,opnd2
Defaults:   RESULT=opnd1,PREC=FFF
Indexable:  opnd1,opnd2,RESULT
```

Operands	Description
opnd1	The name of the variable to which the operation applies. For example, the variables in FSUB A,B correspond to the common algebraic notation A-B. If the RESULT= operand is not specified, then opnd1 is also the implicit result. This operand may not 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 integer constant ('immediate data') between -32768 and +32767 may be specified.
RESULT=	This optional operand can be coded with the name of a variable in which the result is to be placed. In this case, the variable specified by the first operand is not modified.
PREC=	All possible combinations of single and extended precision are permitted. An immediate value for opnd2 will be converted to a single precision value regardless of any other method of precision specification discussed below.

The PREC operand is specified as xyz; where x, y, and z are characters representing the precision of opnd1, opnd2, and the RESULT operands respectively. Either 2 or 3 characters must be specified depending on whether or not the RESULT operand was coded. Permissible characters are:

F = Single-precision (32 bits)
L = Extended-precision (64 bits)
* = Default (single-precision)

If the precision of an operand is not established by the PREC operand, it will default to single precision.

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

Index Registers

The index registers (#1 and #2) may not be used as operands in floating-point operations since they are only 16 bits in length. The registers may, however, be used to specify the address of a floating-point operand.

Return Codes

Floating-point operations produce return codes in the task code word, referred to by taskname (see PROGRAM/TASK). These codes must be tested immediately after the floating-point instruction is executed or the code may be destroyed by subsequent instructions.

Code	Description
-1	Successful completion
1	Floating point overflow
5	Floating point underflow

Example

```
FSUB      L4,F2,PREC=LF

FSUB      L4,2,PREC=L*
```

GETEDIT

Data Formatting

GETEDIT moves data from a terminal or a text buffer, converts the data, and stores it in variables within the program.

Syntax

```
label       GETEDIT  format,text,(list),(format list),
                     ERROR=,ACTION=,SCAN=,SKIP=,LINE=,
                     SPACES=,PROTECT=

Required:   text, (list), and either format
            or (format list)
Defaults:   ACTION=IO,SCAN=FIXED,PROTECT=NO
Indexable:  none
```

Operands	Description
format	The name of a FORMAT statement or the name to be attached to the format list optionally included in this instruction. This statement or list will be used to control the conversion of the data. This operand is required if the program is compiled with $EDXASM.
text	The name of a TEXT statement defining the text buffer. If data is moved from the terminal, this buffer stores the data as an EBCDIC character string before it is converted and moved into the variables.
list	A description of the variables or locations which will contain the desired data. The list will have one of the following forms:

```
((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.
- No data in input (attempt is made to convert the rest)
- 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).
- The error indicators (return codes) are listed in the description of the CONVTD instruction.
- If the ERROR parameter is not coded, then no error indicator is returned to the user.

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.

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 0 (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 incremented by the value specified.

PROTECT= Code PROTECT=YES 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.

Operation GETEDIT scans the input text buffer and converts data according to the FORMAT list, then stores the data in the users program at the locations specified by the data list.

Example

```
         GETEDIT   FM,TEXT1,(A,(B,F),(C,L))
         .
         .

  TEXT1  TEXT      LENGTH=24
  FM     FORMAT    (I4,F6.2,2X,E10.4)
```

The above example will convert the first 4 characters to an integer and store them at A, then convert the next 6 characters to a single-precision floating-point value and store them at B. The next 2 characters are bypassed. The next 10 characters are converted to extended-precision floating-point (due to type specification E) and stored at C.

See Figure 9 on page 166 for an overview of GETEDIT.

Note: $LINK must be used in order to include the formatting routines which are supplied as object modules. Refer to "Data Formatting Instructions" on page 18 for additional information.

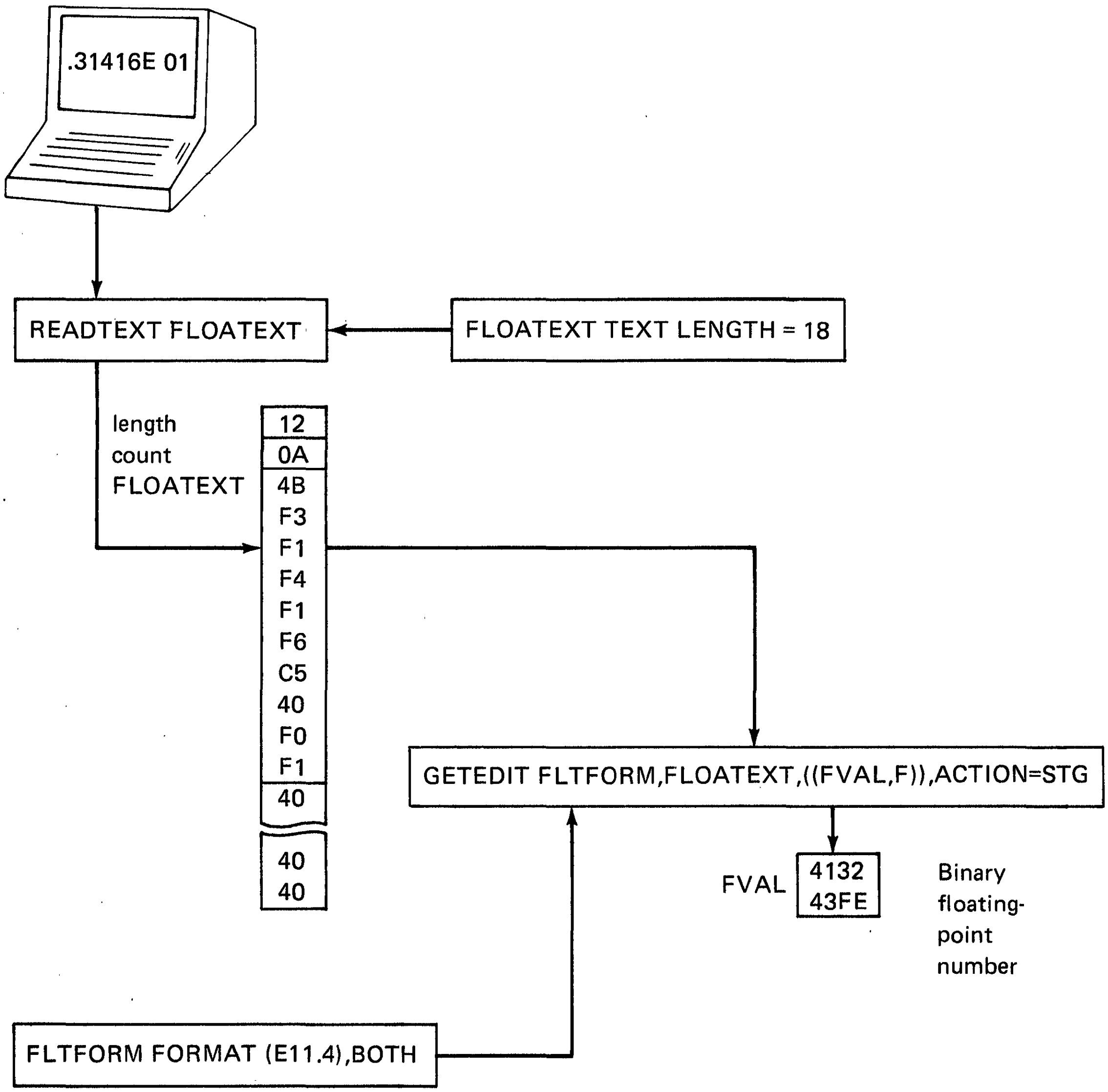


Figure 9. GETEDIT Overview

GETTIME

Timing

GETTIME will cause the contents of the system time-of-day clock to be inserted into a 3-word table in the application program. The 3 words will contain hours, minutes, and seconds, respectively. It is possible to specify that the date is to be stored in an additional 3 words, resulting in a 6-word table containing hours, minutes, seconds, month, day, and year. This instruction is useful when you wish to store the time of day and date with data when it is collected. The maximum time is 23.59.59. At midnight, the time-of-day clock is reset to 0 and the day is incremented by 1.

Note: Day, month and year are incremented and reset as necessary by the Supervisor.

Syntax

```
label        GETTIME loc,DATE=,P1=

Required:    loc
Defaults:    DATE=NO
Indexable:   loc
```

Operands	Description
loc	The address of: (1) a 3-word table in which the time of day will be stored as hours, minutes, and seconds, or (2) a 6-word table in which the time of day and the date will be stored as hours, minutes, seconds, month, day, and year. These numbers are in binary form.
DATE=	Code DATE=YES to obtain the date as well as the time of day. If the system was generated with DATEFMT=DDMMYY on the SYSTEM statement, the TCB code word, $TCBCO, will contain a -2. If DATEFMT=MMDDYY, the code word will be -1. In either case, the table contains month, day, and year in that order. The return code may be used to inform application programs of the standard date format that is desired for each particular system.

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

Example

```
GETTIME   TAB,DATE=YES

TAB  000D  (hours)
     0018  (minutes)
     0005  (seconds)
     000C  (month)
     0019  (day)
     004F  (year)
```

This example is equivalent to 13:24:05, on December 25,1979.

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).

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.

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

Graphics

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.

GOTO

Program Sequencing

GOTO is an unconditional branch to another instruction or a list of instructions from which a selection is made as a function of a specified index value (computed GOTO). The instruction branched to must be on a full-word boundary.

Examples using GOTO are shown under the IF instruction described later in this chapter.

Syntax

```
label       GOTO  loc,P1=

label       GOTO  (loc0,loc1,loc2,...,locn),index,P1=,P2=

Required:   loc
Defaults:   none
Indexable:  index
```

Operands	Description
loc	The address of the instruction to be executed after the unconditional branch. If loc is enclosed in parentheses, the GOTO is indirect and the address of the next instruction is determined by the contents of loc.
loc0	The address of the instruction to be executed if the index value for a computed GOTO is not in the range 1 to n.
loc1,loc2,...,locn	A list of instruction addresses. The address selected will be a function of the value of the index field.
index	The address of an index variable (single-precision value) whose value is to be used to select the target address for the branch. The number of loc instruction addresses +1 must not exceed 50.

GOTO

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

Example

```
GOTO   LOCO                Branch to LOCO
GOTO   (LOC1)              Branch to location address defined
                              by LOC1
GOTO   (ERR,L1,L2),I       Computed GOTO based on value 'I'.
                                If I is 1, branch to L1.
                                If I is 2, branch to L2.
                                Otherwise, branch to ERR.
```

IDCB

EXIO Control

IDCB is used to create a standard immediate device control block.

Syntax

```
label       IDCB     COMMAND=,ADDRESS=,DCB=,DATA=,
                     MOD4=,LEVEL=,IBIT=

Required:   label,COMMAND=,ADDRESS=
Defaults:   LEVEL=1,IBIT=ON
Indexable: not applicable
```

Operands	Description
COMMAND=	The specific I/O operation. Code one of the following keywords shown below. The resulting hexadecimal command code is shown in parentheses. An x represents a character that is filled in by the value specified by MOD4.

READ	- Transfer a byte or word from the device	(0x)
READ1	- Same as READ plus function bit set	(1x)
READID	- Read the device identification word	(20)
RSTATUS	- Read the device status	(2x)
WRITE	- Transfer a byte or word to the device	(4x)
WRITE1	- Same as WRITE plus function bit set	(5x)
PREPARE	- Prepare the device	(60)
CONTROL	- Initiate a control action to the device	(6x)

RESET	- Initiate a device reset operation	(6F)
START	- Initiate a cycle steal operation	(7x)
SCSS	- Initiate a start cycle steal status operation	(7F)

ADDRESS= The device address as two hexadecimal digits.

DCB= The label of a DCB.

DATA= The data word to be transferred to the device by a WRITE, WRITE1, or CONTROL command. Code the actual data as four hexadecimal digits.

MOD4= A four bit device dependent value that modifies the command code specified by the COMMAND code. Code one hexadecimal digit.

LEVEL= The hardware interrupt level to be assigned to the device by a PREPARE command.

IBIT= Code ON or OFF to indicate whether the device is to have the ability to present an interrupt.

Note: Refer to the description manual for the processor in use for more information on IDCBs.

Example

```
IDCB1    IDCB COMMAND=WRITE1,ADDRESS=00,DATA=0041

PREPIDCB IDCB COMMAND=PREPARE,ADDRESS=E4,LEVEL=3,IBIT=ON

WR1IDCB  IDCB COMMAND=START,ADDRESS=E1,DCB=WR1DCB
```

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.

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,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)
```

INTIME

Timing

INTIME is used to provide the user with two forms of interval timing information, reltime and loc. The first, reltime, is a 2-word area in the your program where INTIME will store a value each time an INTIME is executed. This value is equal to the elapsed time since system IPL. This count is expressed in milliseconds and is in double precision integer format. The maximum value for reltime will be reached in approximately 49 days of continuous operation and the counter will then roll over to 0.

The second, loc, is a single-precision integer variable where INTIME will store the time in milliseconds since the previous execution of an INTIME instruction in this task. The maximum interval between calls to INTIME (that is, the maximum value that can be stored at loc) is 65535 milliseconds or 65.535 seconds.

Syntax

```
label       INTIME reltime,loc,INDEX,P2=

Required:   reltime,loc
Defaults:   no indexing
Indexable:  loc
```

Operands	Description
reltime	The address of a 2-word table where a relative time marker may be stored. This field should be defined by DATA 2F'0'. The relative time marker is a double-precision count, in milliseconds, which indicates the relative time at which the last INTIME was issued. It should be initialized to 0. Proper use of this parameter allows you to measure different intervals from the same origin in time.
loc	Buffer address or location where interval time data is to be stored. When reltime = 0, as after initialization, the first interval returned will also be 0.

INDEX — Automatic indexing is to be used. The operand loc must be defined by a BUFFER statement when INDEX is used.

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

Note: Each task in each program in the system has available to it one software driven timer which operates with a precision of 1 millisecond. The STIMER instruction is used to operate this timer in any task.

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.
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. The default is LINSIZE-1.

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 larger 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 of bytes. 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 parameters with respect to an 80-byte line.

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 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=
```

IODEF

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
```

Analog Input

Syntax

```
label        IODEF   AIx,ADDRESS=,POINT=,RANGE=,ZCOR=

Required:    All
Defaults:    RANGE=5V,ZCOR=NO
```

Operands	Description
	Note: The following operand descriptions apply to the five forms of IODEF as previously shown:
PIx	Process Interrupt. Operand x specifies a symbolic reference number used within an application program; range = 1-99. If multiple IODEF PIx statements are included in the program, they must be grouped together.
DOx	Digital Output. Operand x specifies a symbolic reference number used within an application program; range = 1-99. If multiple IODEF DOx statements are included in the program, they must be grouped together.
DIx	Digital Input. Operand x specifies a symbolic reference number used within an application program; range = 1-99. If multiple IODEF DIx statements are included in the program, they must be grouped together.
AOx	Analog Output. Operand x specifies a symbolic reference number used within an application program; range = 1-99. If multiple IODEF AOx statements are included in the program, they must be grouped together.
AIx	Analog Input. Operand x specifies a symbolic reference number used within an application program; range = 1-99. If multiple IODEF AIx statements are included in the program, they must be grouped together.

TYPE=

GROUP — The complete DI/DO/PI group participates in the I/O operation. See SPECPI below if PI is specified as GROUP. DI operates in unlatched mode.

SUBGROUP — A subset of the 16-bit group will be used in the I/O operations. For output operations, all bits not part of the specified subgroup will remain unchanged. For input, the subgroup will be stored right-adjusted in input word with all high order bits set to zero. DI operates in unlatched mode.

BIT — Specified for a user PI bit only (see SPECPI below).

EXTSYNC — Specified when using the hardware external synchronization feature for DI or DO. A count field must be specified on associated SBIO instructions. EXTSYNC also implies latched DI operation mode.

ADDRESS= Specify the two-digit hexadecimal address.

BITS=(u,v) This parameter indicates a portion of a group starting at bit u (u = 0 to 15) for a length v (v = 1 to 16-u). This operand is used only when TYPE=SUBGROUP is specified for DI or DO. Note that it is possible to specify a 16-bit wide subgroup, although it is probably more meaningful in that case to define a normal group and specify a substring in certain I/O cases.

BIT= A number from 0 - 15 specifying the bit to be used for PI.

POINT= Specify the analog output or input point. Point = 0 - 1 for AO, and 0 - 7 for AI relay or 0 - 15 for AI solid state.

RANGE= Specify the range for the multirange amplifier.

```
5V     = 5 VOLTS
500MV  = 500 Millivolts
200MV  = 200     "
100MV  = 100     "
50MV   = 50      "
20MV   = 20      "
10MV   = 10      "
```

ZCOR= This parameter allows the zero correction facility of AI to be used. Specify 'YES' to use zero correction, the default is 'NO'.

SPECPI= Identifies the label of the first instruction of a special process interrupt routine. See SPECPI below.

Example

```
IODEF   PI1,ADDRESS=48,BIT=2
IODEF   PI2,ADDRESS=49,BIT=15
IODEF   DO1,TYPE=GROUP,ADDRESS=4B
IODEF   DO2,TYPE=EXTSYNC,ADDRESS=4A
IODEF   DI1,TYPE=GROUP,ADDRESS=49
IODEF   AI1,ADDRESS=72,POINT=1,RANGE=50MV,ZCOR=YES
IODEF   AO2,ADDRESS=75,POINT=1
```

In this example, two process interrupts are defined, a digital output group, a digital output group as external sync, a digital input group, an analog input point, and an analog output point.

The SBIO instruction is used to reference the digital and analog I/O points as described under the SBIO instruction. Process interrupt are referenced by the POST and WAIT instructions and are described under the respective instruction. Further examples of IODEF statements are shown following the SBIO instruction.

SPECPI - Process Interrupt User Routine

The SPECPI option of the IODEF statement may be used to define a special process interrupt routine. The supervisor will execute a routine written in Series/1 assembler language when the defined interrupt occurs. The purpose is to provide the minimum delay before service of the interrupt, by bypassing the normal supervisor interrupt servicing. Multiple special process interrupt routines are allowed in a program.

TYPE=BIT Control is given to the specified routine when, and only when, an interrupt occurs on the specified bit. On return to the supervisor, the contents of R1 must be the same at entry to the user's routine and R0 must contain either '0' or a POST code. In the latter case, R3 must contain the address of an ECB to be posted by the POST instruction. Register 7 contains the supervisor return address upon entry. If the user routine is in partition 1, the return to the supervisor may be accomplished using BXS (R7). Otherwise return must be made by use of the SPECPIRT instruction. SPECPIRT can also be used in partition 1. The value that is in R7 upon entry may be used to return to the supervisor using BXS (R7) only if the user routine is in partition 1.

TYPE=GROUP Control is given to the specified routine if any bit in the PI group occurs. The user's routine is entered as quickly as possible. The PI group is not read or reset; this is the user routine's responsibility. Return to the supervisor is done with a branch to the entry point SUPEXIT. The module $EDXATSR must be included with the PROGRAM to use SUPEXIT. If interrupt is processed on level 0, the routine may issue a Series/1 hardware exit level instruction (LEX) instead of returning to SUPEXIT. This will improve performance significantly.

Note: Use of TYPE=GROUP requires that you be familiar with the operation of the Series/1 process interrupt feature. Your routine must contain all instructions necessary to read and reset the referenced process interrupt group.

Example using special process interrupt bit

```
IODEF PI2,ADDRESS=48,BIT=3,TYPE=BIT,SPECPI=FASTPI1

FASTPI1   EQU      *

          MVW      R1,SAVER1       SAVE R1
          .
          .
          .
          MVA      PI2,R3          PUT THE ADDRESS OF PI2 IN R3
          MVWI     3,R0            POSTING CODE IN R0
          MVW      SAVER1,R1       RESTORE R1
          SPECPIRT                 RETURN TO SUPERVISOR
```

Example of special process interrupt group

```
          IODEF PI6,ADDRESS=49,TYPE=GROUP,SPECPI=FASTPI2

FASTPI2   EQU       *
```

Control is given to the user at label FASTPI2.

IOR

Data Manipulation

IOR will logical OR operand 2 to operand 1, bit by bit. If either bit is one, the result is a one; if neither bit is one, the result bit will be zero.

Syntax

```
label       IOR     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	This operand identifies the bit string to be ORed with the first operand. 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. 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 (default) DWORD -- Doubleword precision
RESULT=	This operand, which is optional, can 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.

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

Example

```
         IOR    STRING,X'F008',RESULT=ANS
STRING   DATA   X'0F08'     binary 0000 1111 0000 1000
ANS      DATA   F'0' binary zeros
```

After execution of IOR, the variable ANS looks like this:

```
ANS      DATA    X'FF08'     binary 1111 1111 0000 1000
```

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.

Syntax

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

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

Operands	Description
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.

Example: 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.

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 referenced by the task name. 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.

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.

When a LOAD is executed for either an independent program or an overlay, the address of the currently active terminal of the loading program is stored in the program header of the program being loaded.

Syntax

```
label       LOAD  progname,parmname,
                  DS=(dsname1,...,dsname9),EVENT=,
                  LOGMSG=,PART=,ERROR=,STORAGE=,P2=
                or
label       LOAD  PGMx,parmname,DS=(DSx,...),EVENT=,
                  ERROR=,P2=

Required:   progname or PGMx
Defaults:   LOGMSG=YES,STORAGE=0
Indexable:  none
```

Operands	Description
progname	The 1-8 character name of a program stored in an Event Driven Executive library. The user may specify a the volume from which to load the program by separating the program name, 1-8 characters, and the volume name, 1-6 characters, by a comma and enclosing in parentheses, for example, (PROGA,EDX003). The program must reside on disk or diskette.
PGMx	The parameter x is a digit from 1 to 9 specifying which of the overlay programs, defined in the PROGRAM statement, is to be loaded. PGMx is not valid with PART; overlay programs are loaded in space included with the loading program.
parmname	The symbolic label on the first word in a list of consecutive parameter words to be passed to the new program. (See the PROGRAM statement for specification of the length of this list.)

DS= This parameter designates data sets to be passed to the loaded program.

For a non-overlay program load, 1 - 9 data set names can be listed. These names are used to specify data set names at program load time. (See PROGRAM statement.) Data sets may also be specified in the form DSx where x is a digit from 1 to 9 which selects a data set defined in the PROGRAM statement of the loading program. This allows the definition of data sets to be passed to loaded programs to be deferred until the initial load time.

For an overlay program load, specify DSx where x is a digit from 1 to 9 selecting data sets defined in the PROGRAM statement of the loading program.

For example, in a non-overlay situation assume that the PROGRAM statement in the program to be loaded specified a data set list such as:

```
DS=(PARMFILE,??,RESULTS)
```

Then a statement

```
LOAD   progname,parmname,DS=(MYDATA)
```

would yield a final list of

```
DS=(PARMFILE,MYDATA,RESULTS)
```

All unspecified data set names in the program being loaded must be resolved at LOAD time or the operation will not be performed. If a tape data set is passed to another program using the LOAD statement, the loading programs DSCB will be disconnected from the tape data set. This allows the program being loaded to have access to the tape data set using the loading program's DSCB. When the program being loaded completes execution the tape data set will be closed. If the program that issued the LOAD needs to use the tape data set again, it will have to reopen the tape data set using the DSOPEN subroutine or $DISKUT3.

Note: See the PROGRAM statement description for more information on data set specification.

LOGMSG= Specify either YES or NO to indicate whether a "PROGRAM LOADED" message is to be printed on the system logging terminal. The default is YES.

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.

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.

Return Codes

Code	Description
-1	Successful completion
61	The transient loader ($LOADER) is not included in the system
62	In an overlay request, no overlay area exists
63	In an overlay request, the overlay area is in use
64	No space available for the transient loader
65	In an overlay load operation, the number of data sets passed by the LOAD instruction does not equal the number required by the overlay program
66	In an overlay load operation, no parameters were passed to the loaded program
67	A disk or diskette I/O error occurred during the load process
68	Reserved
69	Reserved
70	Not enough main storage available for the program
71	Program not found on the specified volume
72	Disk or diskette I/O error while reading directory
73	Disk or diskette I/O error while reading program header
74	Referenced module is not a program
75	Referenced module is not a data set
76	Data set not found on referenced volume
77	Invalid data set name
78	LOAD instruction did not specify required data set(s)
79	LOAD instruction did not specify required parameters(s)
80	Invalid volume label specified
81	Cross partition LOAD requested, support was not included at sysgen
82	Requested partition number greater than number of partitions in the system

Note: If the program being loaded is a sensor I/O program, and a sensor I/O error is detected, the return code will be a sensor I/O return code, not a load return code.

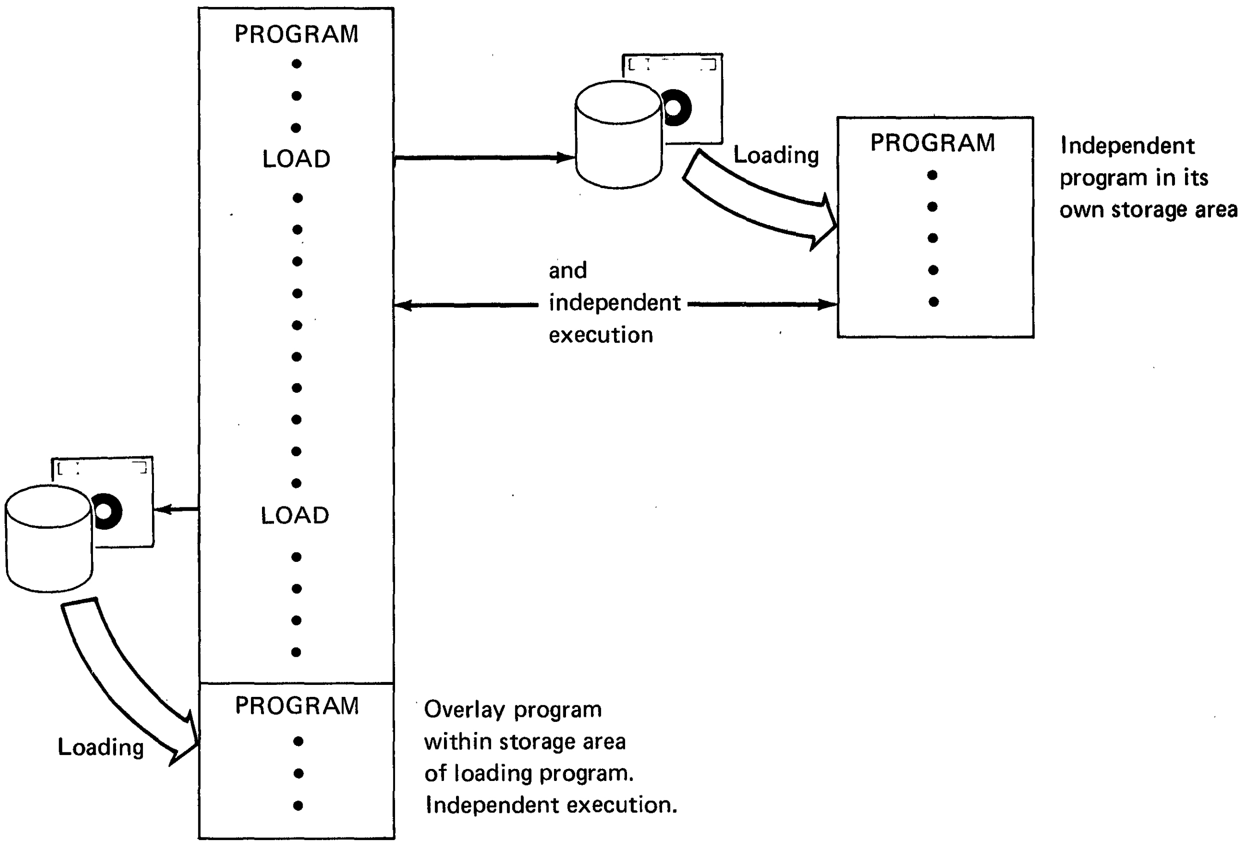


Figure 10. Two Ways of Loading a Program

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.
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.

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 is not indicated by EDX.

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.
```

NEXTQ

Queue Processing

NEXTQ allows the user to add entries to a queue defined by DEFINEQ. A queue element (QE) is removed from the free chain of the queue and placed in the active queue.

Syntax

```
label        NEXTQ   qname,loc,FULL=,P1=,P2=

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

Operands	Description
qname	The name of the queue in which to place the entry. The queue name is the label on the DEFINEQ instruction used to create the queue.
loc	The address of one word of storage which will become an entry in the queue. This might be a single word of data or the address of an associated data area. If loc is coded as '#1' or '#2' then the contents of the selected register will become the entry in the queue.
FULL=	Use this operand to specify the first instruction of the routine to be invoked if a "queue full" 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 NEXTQ 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 full.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Queuing Instructions Programming Example

In the following example all queuing instructions are used. A buffer pool is defined which contains 4 six word buffers. A buffer is obtained, GETTIME is executed and the resulting time is queued. The resulting time is stored in the obtained buffer. When all buffers are allocated, the queue entries are printed on a first-in-first-out basis, then on a last-in-last-out basis, and the buffers used are freed. Each buffer pool/queue instruction is executed 8 times.

```
QTEST     PROGRAM   START
*
* EXAMPLE USING QUEUING INSTRUCTIONS
*
START     FIRSTQ    TIMEBUF,LOC
          IF        (QTEST,EQ,1),GOTO,EMPTY
          GETTIME   *,DATE=YES,P1=LOC
          NEXTQ     TIMEQ1,LOC,FULL=ERROR1
          NEXTQ     TIMEQ2,LOC,FULL=ERROR1
          ADD       CTR,1
          GOTO      START

EMPTY     FIRSTQ    TIMEQ1,OUTADDR1,EMPTY=CHKCTR
          LASTQ     TIMEQ2,OUTADDR2,EMPTY=CHKCTR
          ENQT      $SYSPRTR
          PRINTEXT  SKIP=1
          PRINTNUM  *,6,6,P1=OUTADDR1
          PRINTEXT  SPACES=5
          PRINTNUM  *,6,6,P1=OUTADDR2
          DEQT
          NEXTQ     TIMEBUF,OUTADDR1
          GOTO      EMPTY

CHKCTR    IF        (CTR,GE,8),GOTO,DONE
          GOTO      START
ERROR1    PRINTEXT  '@TIMEQ PREMATURELY FULL@'
DONE      PROGSTOP

*  DATA AREA
TIMEBUF   DEFINEQ   COUNT=4,SIZE=12
TIMEQ1    DEFINEQ   COUNT=10
TIMEQ2    DEFINEQ   COUNT=10
CTR       DATA      F'0'
          ENDPROG
          END
```

NOTE

Disk/Tape I/O

NOTE causes the value of a data set's next-record-pointer, which is maintained by the system, to be stored in your program. The next-record-pointer is the relative record number that will be retrieved by the next sequential READ or WRITE.

Syntax

```
label       NOTE  DSx,loc,P2=

Required:   DSx,loc
Defaults:   none
Indexable:  loc
```

Operands	Description
DSx	Operand x specifies the relative data set number in a list of data sets defined by the user in the PROGRAM statement. The first data set is DS1, the second is DS2, and so on. A DSCB name defined by a DSCB statement may be used in place of DSx.
loc	The address of a full word of storage that will contain the next record pointer, after NOTE is executed. This value can be used as the relative record number (relrecno) in a subsequent POINT or direct READ/WRITE operation.
P2=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

PLOTGIN

Graphics

PLOTGIN provides interactive reading of values of data on curves plotted on screens. The bell is rung and the cross-hair cursor is displayed. The program waits for the user to position the cross-hairs and key any character. That character and the cursor coordinates, scaled by use of the PLOTCB, are obtained for use by the program.

Syntax

```
label       PLOTGIN   x,y,char,pcb,P1=,P2=,P3=,P4=

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

Operands	Description
x,y	Locations for storage of x and y cursor coordinate values.
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.
pcb	Label of an 8-word Plot Control Block.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Plot Control Block

PLOTCB (Plot Control Block) data areas are required by the PLOTGIN, XYPLOT, and YTPLOT instructions.

The plot control block is 8 words of data defined by DATA statements which provide definition of size and position of the plot area on the screen and the data values associated with the edges of the plot area. Indirectly, the scale of the plot is specified. The format of a PLOTCB is:

```
label     DATA     F'xls'
          DATA     F'xrs'
          DATA     F'xlv'
          DATA     F'xrv'
          DATA     F'ybs'
          DATA     F'yts'
          DATA     F'ybv'
          DATA     F'ytv'
```

All 8 explicit values (no addresses) are required and are explained in the text following:

xls: x screen location at left edge of plot area

xrs: x screen location at right edge of plot area

xlv: x data value plotted at left edge of plot

xrv: x data value plotted at right edge of plot

ybs: y screen location at bottom edge of plot

yts: y screen location at top edge of plot

ybv: y data value plotted at bottom edge of plot

ytv: y data value plotted at top edge of plot

POINT

Disk/Tape I/O

POINT causes the value of a data set's next-record-pointer, which is maintained by the system, to be reset to a new value. The system will use this new value in subsequent sequential data set accesses.

Syntax

```
label        POINT DSx,relrecno,P2=

Required:    DSx,relrecno
Defaults:    none
Indexable:   relrecno
```

Operands	Description
DSx	The operand 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. The first data set is DS1, the second is DS2, and so on. A DSCB name defined by a DSCB statement can be substituted for DSx.
relrecno	The new value of the next record pointer, either a constant or the label of the value to be used.
P2=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

POST

Task Control

POST is used to signal the occurrence of an event.

Syntax

```
label       POST  event,code,P1=,P2=

Required:   event
Defaults:   code=-1
Indexable:  event
```

Operands	Description
event	The symbolic name of the event. The name may be defined in an EVENT= operand of another instruction, or with an ECB statement. An explicit ECB must be coded in programs to be compiled with $EDXASM. $S1ASM and the S/370 host assembler both provide automatic generation of the ECB for the event named in the POST instruction. It is not necessary to code an ECB statement with either of these macro assemblers. Process interrupts are special events which may be simulated with a POST. This is useful when one task is waiting for a process interrupt and a second task wishes to activate the first, as in a program termination sequence. In this case, issue a POST PIx where x is a process interrupt number in the range of 1-99 as specified in an IODEF statement.
code	A value, other than zero, to be inserted into the control block for the event. The code word is referred to by the event name and may be used as a flag to indicate a condition or a status.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

POST normally assumes the event is in the same partition as the executing program. However, it is possible to POST an event in another partition using the cross-partition capability of POST. See the System Guide topic on "Cross-Partition Services" for more information.

PRINDATE

Terminal I/O

PRINDATE prints the date on the terminal. The value is printed in the form MM/DD/YY or DD/MM/YY, depending upon the option selected on the SYSTEM statement when the supervisor was generated.

Syntax

```
label       PRINDATE

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

Operands	Description
none	none

PRINT

Listing Control

The PRINT statement is used to control printing of the assembly listing.

A program may contain any number of PRINT statements. One PRINT statement controls the printing of the assembly listing until another is encountered. Each option remains in effect until the corresponding opposite option is specified.

The GEN/NOGEN option is not supported by $EDXASM.

Syntax

```
blank        PRINT   ON/OFF,GEN/NOGEN,DATA/NODATA

Required:    none
Defaults:    (Initially) ON,GEN,NODATA
Indexable:   none
```

Operands	Description
	The operands may be specified in any sequence.
ON	A listing is printed.
OFF	No listing is printed.
GEN	All statements generated by instructions are printed.
NOGEN	Statements generated by instructions are not printed with the exception of MNOTE (error messages) which will print regardless of NOGEN. The instruction itself will still appear in the listing.
DATA	Constants are printed out in full in the listing.
NODATA	Only the leftmost 8 bytes of constants are printed on the listing.

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. 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.

PROTECT= Code PROTECT=YES to write protected characters to a screen device for which this feature is supported (the IBM 4978/4979 display). This operand is meaningful only for STATIC logical screens.

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

Code	Description	
-1	Successful completion	
1	Device not attached	
2	System error (busy condition)	
3	System error (busy after reset)	
4	System error (command reject)	
5	Device not ready	
6	Interface data check	
7	Overrun received	
>10	Codes greater than 10 represent possible multiple errors. To determine the errors, subtract 10 from the code and express the result as an 8-bit binary value. Each bit (numbering from the left) represents an error as follows:	
	Bit	Description
	0	Unused
	1	System error (command reject)
	2	Not used
	3	System error (DCB specification check)
	4	Storage data check
	5	Invalid storage address
	6	Storage protection check
	7	Interface data check

Figure 11. Terminal I/O Return Codes

Note: If for devices supported by IOS2741 (2741, PROC) an error code greater than 128 is returned, subtract 128; the result then contains status word 1 of the ACCA. Refer to the Communications Features Description manual for determination of the special error condition.

PRINTEXT

Example

```
PRINTEXT  TEXT1

PRINTEXT  '@START OF PROGRAM'

PRINTEXT  TEXT2,SPACES=4

PRINTEXT  TEXT3,LINE=1,SKIP=2

PRINTEXT  SKIP=1

PRINTEXT  CODES,XLATE=NO
```

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.

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. 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 since it will be generated automatically.

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 $$EDXVOL. A data set control block (DSCB) is created just as for any other data set name. However, special processing 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 the sequence '$$' is used as a data set name, a DSCB is created but no attempt is made to open the data set. All other data sets are processed in the normal fashion. This is useful for reserving a DSCB in the PROGRAM header so that it can be filled in and opened (using DSOPEN) after execution begins.

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 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. It can also be used if the application program does not wish to use Event Driven Executive data set facilities at all.

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 restrictions as to its contents. If a program is loaded using $L and it has a PARM specification, the parameters will be initiallized to zero.

PGMS= Names of 1-9 programs which may be loaded as overlays during execution of this program. Programs are specified by name only if they reside on the IPL volume or by (name,volume) if they reside elsewhere. The same coding rules apply as for DS above.

Space will be reserved within this program for the largest of the overlay programs identified in this list, thus insuring that space will be available for the overlays when the program is executed. Overlay programs are invoked using LOAD; only one overlay program can be executed at any one time because each one uses the same space. See the description of the LOAD instruction for additional information.

Notes:

1. PGMS can only be coded for a main program and not in the PROGRAM statement of an overlay program.

2. PGMS cannot specify tape data sets.

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.

TERMERR= Specifies the label of a routine which will handle unrecoverable terminal errors. See "Error Handling" on page 44 for a description of the use of this operand.

FLOAT= Specify FLOAT=YES if floating point instructions are used by the initial task.

MAIN= Specify MAIN=NO if this program does not contain the primary task of a program. For example, code MAIN=NO if this program is a subroutine or any other section of a program which is being prepared separately and will later be link-edited to a main program. Such a program is called a subprogram. Link editing of program modules is only possible with the $LINK utility from the Program Preparation Facility, (5719-XX2 or 5719-XX3) or Series/1 macro assembler, (5719-ASA).

Note: Subprograms must not contain TASK, ENDTASK, or ATTNLIST statements.

MAIN=NO suppresses the generation of the Program Header and the Task Control Block, thereby reducing the storage size of the subprogram. If MAIN=NO then none of the other operands of the PROGRAM statement are meaningful.

When a subprogram is to be assembled by $EDXASM the PROGRAM statement may be omitted entirely.

ERRXIT= Specifies the label of a 3 word list which points to a routine which is to receive control if a hardware error or program exception occurs while the primary task is executing. This task error exit routine must be prepared to completely handle any type of program or machine error. See the System Guide section on Task Error Exits before attempting to use the operand.

If the primary task is part of a program which shares resources such as QCBs, ECBs, or Indexed Access Method update records with other programs, it is often necessary to release these resources even though your program cannot continue because of an error. The supervisor does not release resources for you, but the task error exit facility enables you to take whatever action that is appropriate.

The format of the task error exit list is:

WORD 1	The count of the number of parameter words which follow (always F'2')
WORD 2	The address of the user's error exit routine
WORD 3	The address of a 24 byte area in which the Level Status Block (LSB) and Processor Status Word (PSW) from the point of error are placed before the exit routine is entered. Refer to a Series/1 processor description manual for a description of the LSB and PSW.

STORAGE= Specifies in bytes the quantity of additional storage which should be added to the size of the program itself when it is loaded for execution. This provides a dynamic increment of storage at load time. This value may be overridden by a parameter on the LOAD instruction, thus dynamically altering the space available to the program. The address and length of the additional storage is contained in the variables $STORAGE and$LENGTH respectively and may be referenced by your program during execution.

The amount of storage is rounded up to a multiple of 256 bytes. $LENGTH contains the number of 256 byte pages that are available for current execution. If no dynamic area is specified, $LENGTH contains 0 and $STORAGE contains the address of the program's primary task.

Storage can be any value from 0 to 65,535 minus the size of the program itself. If the storage required is not available at LOAD time, the program will not be loaded.

The amount of storage required by a program for such things as buffers, queues, or data often varies depending on its input. Dynamic storage provides a way to adjust the amount of storage available without recompiling your program. The PROGRAM statement can be used to define the amount of dynamic storage for either minimal or typical processing requirements and the LOAD instruction can be used to expand the work space when processing will require more storage. For example, on a daily basis a program may have to read about 1000 bytes of data into storage, analyze it and format it into a report. Once a month it may be required to process

30 days worth of data (30,000 bytes) in the same way. Instead of wasting 29,000 bytes of storage every day, dynamic storage can be used to adjust the size to meet requirements.

WXTRN= Specify WXTRN=NO if WXTRN statements for entry points SVC, SETBUSY, and SUPEXIT are not to be generated by PROGRAM. WXTRN=YES causes the WXTRNs to be created. These entry points must be defined for Series/1 assembler language programs which contain references to them; however the WXTRNs have no effect on programs which do not refer to them and thus the default is WXTRN=YES. The NO option is provided primarily to allow selective use of EXTRN statements for the entry points at the discretion of Series/1 assembler language programmers.

Examples of valid PROGRAM statements

```
TASK1    PROGRAM  START1
```

The primary task is named TASK1 and the first executable instruction has the label START1. The priority of TASK1 is the default priority, 150.

```
TASK2    PROGRAM  BEGIN,300,FLOAT=YES
```

The primary task, which is named TASK2, has a priority 300 and starts at the label BEGIN. Floating point instructions will be used.

```
TASK3    PROGRAM  GOPROG,DS=NAME1
```

The primary task, TASK3, starts at GOPROG. One data set, NAME1, is defined. All disk I/O statements will refer to this data set by the symbolic name DS1.

```
TASK4    PROGRAM  START4,DS=((MYDATA,110011))
```

The primary task, TASK4, starts at START4 and uses one tape data set. That data set is on a standard labeled tape where the VOL1 label contains 110011 as the volume serial number and the HDR1 label contains MYDATA as the data set name. These labels were written using the $TAPEUT1 utility INITIALIZE function.

```
TASK5    PROGRAM  START5,DS=(($$EDXVOL,TU088))
```

The primary task, TASK5, starts at START5 and uses one tape data set. That tape data set is either on a no label tape or bypass label processing is being used and the tape device ID is TU088.

```
TASK6    PROGRAM  START6,DS=(??,(NAME2,EDX002)),
                  PGMS=(OLAY1,OLAY2),STORAGE=1000
```

TASK6 starts at START6. Two data sets are defined. The name of DS1 will be specified at program load time. The second data set, DS2, has the name NAME2 and resides on the logical volume named EDX002. Two overlays are defined, OLAY1 and OLAY2. A 1000-byte area will be appended to the program and its address placed in $STORAGE.

```
TASK7    PROGRAM  START7,DS=(MYDS1,(MYDS2,100001),
         (OUTPUT,??),??)
```

The primary task, TASK7, starts at START7 and uses 4 data sets. MYDS1 is a disk or diskette data set in the IPL volume. MYDS2 is a tape data set on standard labeled tape number 100001. The last two data sets require operator prompting. The third data set will be prompted for as OUTPUT(NAME,VOLUME); the fourth will be prompted as DS4(NAME,VOLUME). Either or both of the latter data sets may be specified by the operator as disk, diskette, or tape data sets.

PROGSTOP

Task Control

PROGSTOP is used to terminate execution of a program and release the storage allocated to it. There can be more than one PROGSTOP statement in a program. You are responsible for ensuring that all other tasks in a program are inactive at the time when the last active task of the program executes a PROGSTOP. The results of executing a PROGSTOP in a program with multiple active tasks are unpredictable.

You are also responsible for assuring that no asynchronous events remain outstanding. If your program contains an ECB for an event that has not yet occurred, you must WAIT on the event before PROGSTOP. The following instructions can generate asynchronous events: READ, WRITE, STIMER, LOAD, ENQ, and ENQT. Also, if your program can be posted by another program, you must WAIT for the POST or prohibit the other program from posting before executing PROGSTOP.

PROGSTOP will perform a close (CONTROL CLSOFF) for any open tape data set that was defined by the PROGRAM statement or passed by another program.

Note that comments cannot be included on a PROGSTOP statement, unless one or both of the allowable operands are included in the instruction.

Syntax

```
label       PROGSTOP   code,LOGMSG=,P1=

Required:   none
Defaults:   code = -1, LOGMSG=YES
Indexable:  none
```

Operands	Description
code	The posting code to be inserted in the EVENT named in the associated LOAD statement.

LOGMSG= Code either YES or NO to indicate whether or not a "PROGRAM ENDED" message is to be typed on the terminal being used by this program.

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

PUTEDIT

Data Formatting

PUTEDIT is used to get data from variables within the program, convert it to a character string, and either store it in an output text buffer or send it to a terminal.

PUTEDIT uses the specified FORMAT statement and the data list and converts and moves the elements one by one into the text buffer.

Syntax

```
label       PUTEDIT  format,text,(list),(format list),
                     ERROR=,ACTION=,SKIP=,LINE=,SPACES=,
                     PROTECT=

Required:   text, (list), and either format
            OR (format list)
Defaults:   ACTION=IO,PROTECT=NO
Indexable:  none
```

Operands	Description
format	The name of a FORMAT statement or the name to be attached to the format list optionally included within this instruction. This statement or list will be used to control the conversion of the data. This operand is required if the program is compiled with $EDXASM.
text	The name of a text statement defining the text buffer. If data is moved to the terminal, this buffer stores the data (as an EBCDIC character string) after it is converted from the variables and before it is sent to the terminal. Note: This TEXT statement must be large enough to contain all the EBCDIC characters generated by this instruction.

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

The error indicators (return codes) follow:

Return Codes

Code	Description
-1	Successful completion
1	No data in field
2	Field omitted
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 (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 screen device for which this feature is supported. (The IBM 4978/4979 display). This operand is meaningful only for STATIC logical screens.

Example

```
         PUTEDIT   FM,TEXT1,(A,(B,F),(C,L))

TEXT1    TEXT      LENGTH=28
FM       FORMAT    (I4/F6.2,2X,'DATA=',E10.4)
```

The above example will convert the integer A into the first 4 positions of TEXT1 followed by a carriage return command. Then, the next 6 positions will contain the variable B followed by 2 spaces. The literal 'DATA=' will then follow with the extended precision variable C converted into the last 10 positions.

Note: $LINK must be used in order to include the formatting routines which are supplied as object modules. Refer to "Data Formatting Instructions" on page 18 for additional information.

QCB

Task Control

QCB generates a five-word Queue Control Block (QCB) for use with the ENQ and DEQ instructions.

Normally this statement will not be needed in application programs if the program is to be assembled by the Host or Series/1 macro assemblers. In this case queue control blocks are automatically generated for the user as a consequence of naming a resource in a DEQ instruction. However, it may be used for special purposes such as controlling their location within a program. The user must explicitly code any necessary QCBs in programs that are to be compiled by $EDXASM.

A maximum of 25 QCB statements may be coded in a program. If more than 25 QCBs are required, they must be coded using the DATA statement. For example:

```
QCB1      QCB
```

is equivalent to coding,

```
QCB1      DATA      F'-1'
          DATA      2F'0'
          DATA      2F'0'
```

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

<u>Syntax</u>

```
label        QCB   code

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

Operands	Description
label	The label of the QCB statement is used as the name of the resource it represents. It is used as an operand in ENQ and DEQ instructions.
code	Initial value of the code field (word 1). If this word is non-zero, the resource whose usage is controlled by this QCB is defined as not in use.

QUESTION

Terminal I/O

QUESTION allows the terminal operator to choose the direction of a conditional branch in the program. The prompt message (normally in the form of a question) is printed unconditionally, after which the operator may enter Y (or any string beginning with Y) for yes, or N (or any string beginning with N) for no. Note that advance input may accompany the response. If an invalid response is entered, the operator is prompted until a Y or N is entered. The QUESTION instruction must be issued only to terminals which have input capability for response to the prompt.

Syntax

```
label        QUESTION  pmsg,YES=,NO=,SKIP=,LINE=,
                       SPACES=,P1=

Required:    pmsg and either YES= or NO=
Defaults:    If either YES or NO is not specified,
             the corresponding response (Y or N)
             will cause the next instruction to be
             executed.
Indexable:   pmsg,SKIP,LINE,SPACES
```

Operands	Description
pmsg	The prompt message, specified either as the name of a TEXT statement or as an explicit message enclosed in quotes.
YES=	Label of the command at which execution will continue if the answer is YES.
NO=	The label at which execution will continue if the answer is NO.
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 (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.

Syntax

```
label       RDCURSOR  line,indent

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

Operands	Description
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.

Example

```
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.

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.

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

END= Use this 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 data sets, if END is not coded, reading a tapemark will also be treated as an error. The physical position of the tape, under this condition, is the read/write head position is immediately following the tapemark. See CONTROL close functions for repositioning of the data set. Remember also that the count field might not be decremented to zero.

ERROR= Use this operand to specify the first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If this operand is not specified, control will be returned to the next instruction after the READ and you must test the return code in the task code word for errors. This operand must not be used if WAIT=NO is coded.

WAIT= If this operand is allowed to default or if it is coded as WAIT=YES, the current task will be suspended until the operation is complete.

If the operand is coded as WAIT=NO, control will be returned after the operation is initiated and a subsequent WAIT DSx must be issued in order to determine when the operation is complete.

END and ERROR cannot be coded if WAIT=NO is coded. You must subsequently test the return code in the Event Control Block (ECB) named DSx or in the task code word (referred to by 'taskname'). Two codes are of special significance. A -1 indicates a successful end of operation. A +10 indicates an 'End of Data Set' and may be of logical significance to the program rather than being an error. For programming purposes, any other return codes should be treated as errors.

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

READ normally assumes the buffer (loc operand) is in the same partition as the currently executing program. However, a READ into a buffer in another partition is possible using the cross-partition capability of READ. See the System Guide topic on "Cross-Partition Services" for more information.

Disk/Tape Return Codes

Disk/tape I/O return codes are returned in two places:

- The first word of the DSCB (either DSn or DSCB name) 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 for disk and tape are shown in tables later in this section.

Following an error condition on tape, the read/write head position is immediately following the error record. The error retry has been attempted, but was unsuccessful. The count field may or may not have been decremented to zero under this condition.

If detailed information concerning an error is desired, it may be obtained by printing all or part of the contents of the disk data blocks (DDBs) or tape data blocks (TDBs), located in the supervisor area of partition 1. This can be accomplished in either of two ways: (a) by using the $LOG utility (see System Guide for details of use), or (b) by using the following information. The starting address of the DDBs/TDBs may be obtained from the link-edit map of the supervisor. DDBs/TDBs can also be located by the field $DISKDDB in the communications vector table (CVT). Use the PROGEQU equate table to reference $DISKDDB, DDBEQU equate table for DDB, and the TDBEQU equate table for the TDB fields. The contents of the DDBs and the TDBs are described in the IBM Series/1 Event Driven Executive Internal Design, LY34-0168, under the headings of 'Disk Data Block', 'DDB Equates'. 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/TDB in storage.

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.

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.

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).

Code MODE=LINE if the string to be read can include imbedded blanks.

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 operation 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.

Note: 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. 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 increased by the value specified.

Return Codes

Code	Description
-1	Successful completion
1	Device not attached
2	System error (busy condition)
3	System error (busy after reset)
4	System error (command reject)
5	Device not ready
6	Interface data check
7	Overrun received
>10	Codes greater than 10 represent possible multiple errors. To determine the errors, subtract 10 from the code and express the result as an 8-bit binary value. Each bit (numbering from the left) represents an error as follows:

Bit	Description
0	Unused
1	System error (command reject)
2	Not used
3	System error (DCB specification check)
4	Storage data check
5	Invalid storage address
6	Storage protection check
7	Interface data check

Figure 13. Terminal I/O Return Codes

Note: If for devices supported by IOS2741 (2741, PROC) an error code greater than 128 is returned, subtract 128; the result then contains status word 1 of the ACCA. Refer to the Communications Features Description manual for determination of the special error condition.

Value	Transmit	Receive
x'8Fnn'	NA	LINE=nn received
x'8Enn'	NA	SKIP=nn received
-2	NA	Line received (no CR)
-1	Successful completion	New line received
1	Not attached	Not attached
5	Disconnect	Disconnect
8	Break	Break

Figure 14. Virtual Terminal Communication Return Codes

Following is a further description of the above values for a receive operation:

LINE=nn (x'8Fnn'): This code is posted for READTEXT or GETVALUE instructions if the other side sent the LINE forms control operation; it is transmitted so that the receiving program may reproduce on a real terminal (for printer spooling applications for example) the output format intended by the sending program.

SKIP=nn (x'8Enn'): The sending program transmitted SKIP=nn.

Line Received (-2): This code indicates that the sending program did not send a new line indication, but that the line was transmitted because of execution of a control operation or a transition to the read state. This is how, for example, a prompt message is usually transmitted with READTEXT or GETVALUE.

New Line Received (-1): This code indicates transmission of the carriage return at the end of the data. The distinction between a new line transmission and a simple line transmission is, again, made only to allow preservation of the original output format.

Not attached (1): If the virtual terminal accessed for the operation does not reference another virtual terminal, then this code is returned.

Disconnect (5): This code value corresponds to the not-ready indication for real terminals; its specific meaning for virtual terminals is that the program at the other end of the channel terminated either through PROGSTOP or operator intervention.

Break (8): The break code indicates that the other side of the channel is in a state (transmit or receive) which is incompatible with the attempted operation. If only one end of the channel is defined with SYNC=YES (on the TERMINAL statement), then the task on that end will always receive the break code, whether or not it attempted the operation first. If both ends are defined with SYNC=YES, then the code will be posted to the task which last attempted the operation. The break code may thus be understood as follows: when reading (READTEXT or GETVALUE), the other program has stopped sending and is waiting for input; when writing (PRINTEXT or PRINTNUM), the other program is also attempting to write. Note that current Event Driven Executive programs, or future programs, which do not interpret the break code, must always communicate through a virtual terminal which is defined with SYNC=NO (the default).

<u>Example</u>

```
           READTEXT  OPTION,'ENTER OPTION: ',PROMPT=COND
           READTEXT  NAME,'ENTER YOUR NAME: '
           READTEXT  PASSWORD,'ENTER PASSWORD: ',PROTECT=YES
           READTEXT  NEXTLINE,MODE=LINE
               .
               .
OPTION     TEXT      LENGTH=2
NAME       TEXT      LENGTH=44
PASSWORD   TEXT      LENGTH=8
NEXTLINE   TEXT      LENGTH=80
```

RESET

Task Control

RESET is used to reset or clear an event or a Process Interrupt.

When an event occurs for which a task is waiting, the task will again become active. If the task were subsequently to issue another WAIT instruction for the same event, without taking any special action, the event is still defined as having occurred and no wait would be performed. It is necessary to define the event as not occurred in order to cause a new wait. This is the function of the RESET instruction.

The RESET instruction need not be used for the event defined by the EVENT operand of either a PROGRAM or a TASK statement. RESET must not be used for this event prior to executing the ATTACH instruction, since RESET will cause the ATTACH to operate as though the task were already attached.

Events are named logical entities which are represented in storage by a system control block called an Event Control Block (ECB). Resetting an event is physically done by setting the first word of its ECB to 0.

Syntax

```
label       RESET   event,P1=

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

Operands	Description
event	The symbolic name of the event being reset. For process interrupt, use PIx, where x is a user process interrupt number in the range 1-99.
P1=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

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.

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.

ERROR= ERROR= specifies the label of the instruction to be executed if the SBIO instruction is unsuccessful after two retries. If ERROR is not coded, execution will proceed sequentially. In either case, the task code word, whose implicit label is the task name, will contain the return code. The return codes are shown later in this section.

EOB= EOB= may be specified for buffer operations with automatic indexing. A branch is taken to the specified label under two conditions. In the first case, if the last element of the buffer is used during execution of the SBIO, the branch will be taken with a return code of $OK in the task name. Secondly, if the buffer is either full (AI/DI) or logically empty (AO/DO) when the SBIO is executed, the branch will be taken without executing the SBIO and a code of $BFRPFE will be in the task name. In either case, the buffer count is not reset. This is the user's responsibility. (See 'Return Codes' in this section)

INDEX A keyword used to specify that automatic indexing (incrementing of the effective address) of the defined BUFFER is to be performed as part of the execution of this SBIO.

BITS= BITS=(u,v) is used to specify the portion of a digital group or subgroup defined in the referenced IODEF, to be used in an I/O operation. BITS= may not be used with either AI, AO, DO PULSE or external sync DI/DO operations. u is the starting bit number (0-15) relative to the start of the defined group or subgroup. v is the length of the bit string (=1 to 16-u, or as limited by the IODEF subgroup definition).

Return Codes

The task name is the label of a location which will contain a return code after a sensor based I/O operation. These codes should be referenced by the symbolic names shown in the return code table which follows, instead of by an absolute number, to allow future programming flexibility. If any sensor I/O is used, these labels are automatically defined.

Code	EQU	Description
-1	$OK	Command successful
90	$DNA	Device not attached
91	$DNU	Busy or in exclusive use
92	$BAR	Busy after RESET
93	$CMDREJ	Command reject
94	$INVREQ	Invalid request
95	$IDC	Interface data check
96	$CTLBSY	Controller busy
97	$OVRVOLT	AI over voltage
98	$INVRG	AI invalid range
100	$INVCHA	AI invalid channel (point)
101	$INVCNT	Invalid count field (AI/DI/DO count)
102	$BFRPFE	Buffer previously full or empty (indexing)
104	$DCMDREJ	Delayed command reject

For example:

```
        SBIO AI1,ERROR=AIERR
          :
          :
          :
AIERR   IF   (taskname,EQ,+$OVRVOLT),GOTO,REDO
```

If AI1 is over voltage go to label REDO.
Note that the use of '+' when referencing equated values is necessary for proper assembler operation.

Analog Input

Syntax

```
label       SBIO    AIx,P1=
 or
label       SBIO    AIx,loc,P1=,P2=
 or
label       SBIO    AIx,loc,INDEX,EOB=,P1=,P2=
 or
label       SBIO    AIx,loc,op3,SEQ=YES,P1=,P2=,P3=

Required:   AIx
Defaults:   no indexing, SEQ=NO
Indexable:  loc
```

Operands	Description
AIx	Analog input symbolic reference number defined in an IODEF statement and the label of a single data storage location if loc is not specified.
loc	Buffer address or location where analog input value is to be stored, if required.
op3	If op3 equals INDEX then automatic indexing is used. If op3 is a label or constant then AI sequential read is used.
SEQ=NO	op3 is the number of times to repeat same point.
SEQ=YES	op3 is the number of consecutive AI points.

The input voltage converted by the analog-to-digital converter (ADC) is represented in a 16-bit data word by 11 binary bits plus a sign bit, depending on the amplifier range selected. Bits 13 - 15 of this word is the binary number representing the range of the AI reading. Bit 12 will be zero.

Note: Refer to the 4982 Sensor I/O Description manual, for a detailed discussion of the analog-to-digital conversion.

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

Example: SBIO instructions and IODEF statements for Read Analog Input

```
IODEF AI1,ADDRESS=72,POINT=5

SBIO  AI1                 DATA INTO LOCATION AI1
SBIO  AI1,DAT             DATA INTO LOCATION DAT
SBIO  AI1,BUF,INDEX       AI1 INTO NEXT LOC OF 'BUF'
SBIO  AI1,(BUF,#1)        AI1 INTO LOCATION (BUF,#1)
SBIO  AI1,BUF,2,SEQ=YES   READ 2 SEQUENTIAL AI PTS INTO
                          NEXT 2 LOCATIONS OF 'BUF'
SBIO  AI1,BUF,2           READ THE SAME POINT TWO TIMES
 or                       AND PUT INFORMATION IN TWO
SBIO  AI1,BUF,2,SEQ=NO    LOCATIONS OF BUFF
```

Analog Output

Syntax

```
label       SBIO   AOx,P1=
 or
label       SBIO   AOx,loc,P1=,P2=
 or
label       SBIO   AOx,loc,INDEX,EOB=,P1=,P2=

Required:   AOx
Defaults:   no indexing
Indexable:  loc
```

Operands	Description
AOx	Analog output symbolic reference number defined in an IODEF statement and the label of a single data storage location if loc is not specified.
loc	An explicit constant or an address of the location of the output data, if required.

op3 If op3 equal INDEX then automatic indexing is used.

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

Example: SBIO instructions and IODEF statements for Write Analog Output

```
  IODEF A01,ADDRESS=63

  SBIO  A01                   SET A01 TO VALUE IN 'A01'
  SBIO  A01,DATA              SET A01 TO VALUE IN 'DATA'
  SBIO  A01,1000              SET A01 TO 1000
  SBIO  A01,(0,#1)            SET A01 TO VALUE IN (0,#1)
  SBIO  A01,BUF,INDEX         SET A01 TO VALUE IN NEXT
```

Digital Input

Syntax

```
label        SBIO    DIx,P1=
 or
label        SBIO    DIx,loc,P1=,P2=
 or
label        SBIO    DIx,loc,INDEX,EOB=,P1=,P2=
 or
label        SBIO    DIx,loc,BITS=(u,v),LSB=,P1=,P2=
 or
label        SBIO    DIx,loc,op3,P1=,P2=,P3=

Required:    DIx
Defaults:    no indexing,LSB=15
Indexable:   loc
```

Operands Description

DIx Digital input symbolic reference number defined in an IODEF statement and the label of a single data storage location if loc is not specified.

loc — Buffer address or location where digital input value is to be stored.

op3 — If op3 = INDEX, automatic indexing is used.

If op3 is the label of a variable or a constant representing the count of external synchronization read cycles, external synchronization is implied and EXTSYNC must have been specified in the associated IODEF statement. This form also provides a latched DI operation. The entire 16-bit group is read.

If EXTSYNC was specified but op3 is not, then a single unsynchronized I/O operation is performed.

BITS=(u,v)

This parameter indicates that the value of a portion of a DI group is to be read starting at bit u for a length v. Bits are numbered from 0 - 15. Bit u is the relative bit number starting at 0, within the group or subgroup defined in the IODEF statement.

LSB= — This parameter may only be used if BITS= is specified in the SBIO statement. It defaults to bit 15. Input data will be right justified to this bit with all unused bits set to 0.

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

Example: SBIO instructions and IODEF statements for Read Digital Input:

```
IODEF DI1,TYPE=GROUP,ADDRESS=49
IODEF DI2,TYPE=SUBGROUP,ADDRESS=48,BITS=(7,3)
IODEF DI3,TYPE=EXTSYNC,ADDRESS=62

SBIO  DI1                     DATA INTO LOC 'DI1'
SBIO  DI1,DATA                DI1 INTO LOC 'DATA'
SBIO  DI1,(0,#1)              DI1 INTO LOC (0,#1)
SBIO  DI1,BUF,INDEX           DI1 INTO NEXT LOC OF 'BUF'
SBIO  DI1,BDAT,BITS=(3,5)     BITS 3 TO 7 OF DI1 INTO 'BDAT'

SBIO  DI2                     BITS 7-9 OF DI2 INTO 'DI2'
SBIO  DI2,DAT2                BITS 7 TO 9 OF DI2 INTO 'DAT2'
SBIO  DI2,D,BITS=(0,3)        BITS 7-9 OF DI2 INTO 'D'
SBIO  DI2,E,BITS=(0,1)        BIT 7 OF DI2 INTO 'E'
SBIO  DI2,F,BITS=(2,1),LSB=7 BIT 9 OF DI2 INTO
                                LOCATION F BIT 7
SBIO  DI3,G,128                READ 128 WORDS INTO 'G'
                                USING EXTERNAL SYNC
```

Digital Output

Syntax

```
label       SBIO   DOx,P1=
 or
label       SBIO   DOx,loc,P1=,P2=
 or
label       SBIO   DOx,loc,INDEX,EOB=,P1=,P2=
 or
label       SBIO   DOx,loc,BITS=(u,v),LSB=,P1=,P2=
 or
label       SBIO   DOx,loc,op3,P1=,P2=,P3=
 or
label       SBIO   DOx,(PULSE,dir)

Required:   DOx
Defaults:   no indexing,LSB=15
Indexable:  loc
```

Operands   Description

DOx — Digital output symbolic reference number defined in an IODEF statement and the label of a single data storage location if loc is not specified.

loc — An explicit constant or an address where data to be written is stored. Data must be right justified.

op3 — If op3 equal INDEX then automatic indexing is used. If op3 is a label or constant then external sync is used.

BITS=(u,v) — This parameter indicates that the specified value is to be written into a portion of the DO group starting at bit u for a length of v, without affecting the condition of the other bits of the same group. Bits are numbered from 0 - 15. Bit u is the relative bit number (starting at 0, within the group or subgroup defined in the referenced IODEF statement.

LSB= — This parameter may only be used if BITS= is coded on the SBIO statement. It defaults to bit 15. Output data will be taken from the output word with this bit being the least significant bit.

(PULSE,dir) — Specifies a pulse is to be generated on the associated digital output group or subgroup. Directions accepted are ON (or UP) and OFF (or DOWN).

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

Example: SBIO instructions and IODEF statements:

Write Digital Output

```
IODEF DO3,TYPE=GROUP,ADDRESS=4B
IODEF DO12,TYPE=SUBGROUP,ADDRESS=4A,BITS=(5,4)
IODEF DO13,TYPE=EXTSYNC,ADDRESS=4F

SBIO  DO3                    VALUE OF LOCATION 'DO3' to DO3
SBIO  DO3,DODATA             VALUE OF 'DODATA' TO DO3
SBIO  DO3,1023               SET DO3 TO 1023
SBIO  DO3,(DATA,#1)          VALUE AT (DATA,#1) TO DO3
SBIO  DO3,7,BITS=(3,3)       SET BITS 3 TO 5 OF DO3 TO 7

SBIO  DO12,15                SET BITS 5 TO 8 OF DO12 TO 15
SBIO  DO12,X,BITS=(0,4),     SET BITS 5 TO 8 OF DO12
                                 TO VALUE IN 'X'
SBIO  DO12,1,BITS=(0,1)      SET BIT 5 OF DO12 TO 1
SBIO  DO13,Y,80              WRITE 80 LOCATIONS OF 'Y'
                               TO DO13 EXTERNAL SYNC
```

Example: Pulse Digital Output:

```
IODEF DO13,TYPE=SUBGROUP,BITS=(3,1)
IODEF DO14,TYPE=SUBGROUP,BITS=(7,4)

SBIO  DO13,(PULSE,UP)        PULSE DO13 BIT 3 TO ON
                               AND THEN OFF
SBIO  DO14,(PULSE,DOWN)      PULSE DO14 BITS 7-10
                               OFF AND THEN ON
```

SCREEN

Graphics

SCREEN converts x and y numbers representing a point on the screen of a terminal to the 4-character text string which will be interpreted by the terminal as the graphic address of the point. The length of the text string is set to 5 if CONCAT=NO and ENHGR=YES. The length of the text string is set to 4 if CONCAT=NO and ENHGR=NO. Used with CONCAT, this instruction can build a graphical message to the terminal.

Syntax

```
label        SCREEN   text,x,y,CONCAT=,ENHGR=,P1=,P2=,P3=

Required:    text,x,y
Defaults:    CONCAT=NO,ENHGR=NO
Indexable:   none
```

Operands	Description
text	Location of text string at least 4 characters long.
x,y	Screen coordinates of point to be translated. Range is 0 to 1023 for full width of the screen and 0 to 779 for the screen height. Operands x and y may be locations containing data or explicit values, but both must be of the same type. Refer to ENHGR below for enhanced range of 0 to 4086.
CONCAT=	YES - Allows the concatenation of this conversion to whatever is already in text. The text string length is modified plus 4 or (plus 5 if ENHGR=YES is coded).
ENHGR=	YES - Extends the range to 0 to 4095 for full width of the screen and 0 to 3120 for the screen height. When coded YES, a 5 character graphical instruction is compiled.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

SHIFTL

Data Manipulation

Contents of operand 1 are shifted left by the number of bit positions specified by operand 2. Vacated positions on the right are filled with zeroes. If operand 2 is a variable, it is assumed to be single-precision, and the shift count is its value. If the value exceeds the precision in bits, of operand1, the value is divided by the precision and the remainder is used in place of the original value.

Syntax

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

Required:    opnd1,opnd2
Defaults:    count=1,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	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 in opnd1 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

SHIFTL

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.

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

Example

```
SHIFTL A,2                  SHIFT A LEFT 2 BIT POSITIONS
```

SHIFTR

Data Manipulation

Contents of operand 1 are shifted right by the number of bit positions specified by operand 2. Vacated positions on the left are filled with zeros. If operand 2 is a variable it is assumed to be single-precision, and the shift count is its value. If the value exceeds the precision in bits, of operand1, the value is divided by the precision and the remainder is used in place of the original value.

Syntax

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

Required:   opnd1,opnd2
Defaults:   count=1,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	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 in opnd1 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

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.

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

Example

```
SHIFTR C,24,DWORD,RESULT=E    SHIFT C RIGHT 24 BITS,
                              STORE RESULT AT E
```

SPACE

Listing Control

The SPACE statement is used to insert one or more blank lines in the listing.

Syntax

```
blank        SPACE value

Required:    none
Defaults:    value = 1
```

Operands	Description
value	A decimal value specifying the number of blank lines to be inserted. If no value is entered, one blank will be inserted. If this value exceeds the number of lines remaining on the page then the statement will have the same effect as an EJECT statement.

SPECPIRT

SPECPIRT is used to return to the supervisor from a special process interrupt (SPECPI) routine. If the user routine is in partition 1, a branch instruction is used to return. Return from another partition requires execution of a Series/1 assembler SELB instruction after registers R0 and R3 are saved in the level block to be selected. SPECPIRT is used only for TYPE=BIT SPECPI routines. See the description of IODEF (SPECPI) for additional information.

Syntax

```
label        SPECPIRT

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

Operands	Description
none	none

SQRT

Data Manipulation

This instruction is used to find the square root of a double precision integer variable. The instruction is implemented through the USER instruction facility. It is not included in the supervisor. Implementation of this instruction is described further in the Utilities, Operator Commands, Program Preparation, Messages and Codes as an example of how the user may add new instructions to the Event Driven Executive instruction set. If the program is assembled with $EDXASM, $LINK must be used to include the SQRT object module ($$SQRT). The autocall feature of $LINK may be used. For details on the use of the autocall feature, see the Utilities, Operator Commands, Program Preparation, Messages and Codes.

Syntax

```
label       SQRT    rsq,root,rem,P1=,P2=,P3=

Required:   rsq,root,rem
Defaults:   none
Indexable:  none
```

Operands	Description
rsq	The name of a double precision integer that the square root routine is to use. This value must be between 0 and 1,073,741,823 inclusive.
root	The name of a single precision integer where the square root is to be stored.
rem	The name of a single precision integer where the remainder is to be stored.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

STATUS

Data Definition

STATUS is used to define the fields required for referencing a record in the "System Status Data Set" on the host computer.

A STATUS statement is referenced by the TP SET, TP FETCH, and TP RELEASE instructions. See 'Host Communications', in the Communications and Terminal Applications Guide for a description of these instructions and the System Guide for a description of the "System Status Data Set".

Syntax

```
label       STATUS index,key,length,P1=,P2=,P3=

Required:   label,index,key
Defaults:   length=0
Indexable:  none
```

Operands	Description
index	A 1 - 8 alphameric character string. This defines an index in the status data set. One or more entries may be associated with this index, each with a unique key field. It is suggested that a unique index be specified for each Series/1, but this is not a requirement.
key	A 1 - 8 alphanumeric character string. The index and key together define a unique status data set entry. A different key might be used for each application program on a Series/1 which communicates to a host.
length	Specifies the length of an optional buffer to be used in the SET, FETCH, and RELEASE functions of the TP instruction. The maximum buffer length, which may be specified in bytes, is 256. If this operand is omitted, no buffer is defined. If a buffer is specified with a length greater than 0, then it may be named by using

the Px= operand.

The contents of the buffer can be stored in the System Status data set with a TP SET instruction. For a TP FETCH or TP RELEASE, this buffer will serve as an input area.

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

STIMER

Timing

STIMER is used to start a software timer and provide an interrupt after the specified number of milliseconds have elapsed. It allows a means of periodically executing a portion of the user task or providing program delays. The minimum timer setting is 1 millisecond and the maximum setting is 60,000 milliseconds or 60 seconds.

Note: When using a model 4952 or 4953 Processor the minimum setting should not be less than 3 milliseconds.

STIMER may be used in conjunction with the WAIT instruction.

Two STIMER instructions without an intervening WAIT will cause the time interval specified by the first STIMER to be replaced by the interval specified by the second STIMER.

Syntax

```
label       STIMER count,WAIT,P1=

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

Operands	Description
count	The address of a word, or an explicit constant, which specifies the timer setting in milliseconds. The value is an unsigned, 16 bit integer.
WAIT	Specifies that control will not return to the next instruction until the time interval has elapsed. If WAIT is not specified, then a subsequent WAIT instruction must be issued with the keyword 'TIMER' specified as the event being waited upon.
P1=	Parameter naming operand. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further 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. If the subroutine is 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.

SUBTRACT

Data Manipulation

Signed subtraction of operand 2 from operand 1. May be abbreviated SUB.

Note: An overflow condition is not indicated by EDX.

Syntax

```
label       SUBTRACT  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 in opnd1 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 subtract 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

```
SUB    A,B           single-precision subtract
SUB    A,(2,#2)      subtract data at (2,#2) from A
```

TASK

Task Control

The TASK statement defines the beginning of a block of instructions which will execute asynchronously with the attaching task, (and other tasks in the system), according to its assigned priority.

Note: TASK statements may only be coded within main programs, not within subprograms which will later be link edited to a main program.

Each task in a program, except the initial task, begins with a TASK statement and must be terminated with an ENDTASK. The initial task begins with the PROGRAM statement and is terminated by ENDPROG.

Syntax

```
taskname    TASK     start,priority,EVENT=,TERMERR=,FLOAT=,
                     ERRXIT=

Required:   taskname,start
Defaults:   priority=150,FLOAT=NO
Indexable:  none
```

Operands	Description
taskname	The name of the task. A system control block is generated for each task in an application program. This is known as the task control block (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 special significance in program operation. For example, in I/O operations, it is used to store 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 when the task is first attached.

priority — The priority to be assigned to the task. The range is from 1 (highest priority) to 510 (lowest priority). Tasks with priorities 1-255 are run in foreground (Interrupt Level 2) and those with 256-510 are run in background (Interrupt Level 3).

Priorities separate tasks according to their real time needs for processor time. Priority assignments must therefore account for other programs expected to be executing simultaneously

EVENT= — Name of an end event. This event will be posted complete at the termination of this task. The attaching task can, if desired, synchronize its operation by issuing a WAIT for this event. This event name must not be defined explicitly by an ECB since it will be generated automatically.

TERMERR= — See "Error Handling" on page 44 for a description of the use of this operand.

FLOAT= — Specify FLOAT=YES if floating-point instructions are used by this task.

ERRXIT= — Specifies the label of a 3 word list which points to a routine which is to receive control if a hardware error or program exception occurs while the primary task is executing. This task error exit routine must be prepared to completely handle any type of program or machine error. See the System Guide section on Task Error Exits before attempting to use the operand.

If the primary task is part of a program which shares resources such as QCBs, ECBs, or Indexed Access Method update records with other programs, it is often necessary to release these resources even though your program cannot continue because of an error. The supervisor does not release resources for you, but the task error exit facility enables you to take whatever action that is appropriate.

The format of the task error exit list is:

WORD 1 — The count of the number of parameter words which follow (always F'2')

WORD 2 — The address of the user's error exit routine

WORD 3 — The address of a 24 byte area in which the Level Status Block (LSB) and Processor Status Word (PSW) from the point of error are placed before the exit routine is entered. Refer to a Series/1 processor description manual for a description of the LSB and PSW.

<u>Example</u>: See "Example 7: A Two Task Program With ATTNLIST" on page 395 for TASK coding example.

TERMCTRL

Terminal I/O

The TERMCTRL instruction is used to request execution of special terminal control functions. These functions are generally device dependent; the form of the instruction depends on the device.

TERMCTRL may be used with the following device types:

Device	Examples of Functions
2741 Communications Terminal	Set the attention function
4013 Graphics Terminal	Set the attention function
4973 Printer	Set lines per inch
4974 Printer	Set the 4974 control storage
4978 Display	Write or read 4978 storage Blink the cursor Blank the screen
4979 Display	Blank the screen Lock/unlock the keyboard
ACCA devices	Control the modem
Teletypewriter equivalent devices	Write buffered output
Virtual terminals (DEVICE=VIRT)	Set the attention function Pass function codes
Other processors (DEVICE=PROC)	Same as Virtual terminals

The syntax of TERMCTRL, by device, is as follows:

2741 Communications Terminal

Syntax

```
label       TERMCTRL   function,ATTN=

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

Operands	Description	
function:		
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
	DISPLAY	Causes any buffered output to be written to the 2741.

ATTN= NO, to disable the attention function.

YES, to enable the attention function.

This operand must be used in conjunction with the SET function.

Examples:

```
SETATTN     TERMCTRL   SET,ATTN=YES
WRITEPTR    TERMCTRL   DISPLAY
```

TERMCTRL

4013 Graphics Terminal

Syntax

```
label       TERMCTRL  function,ATTN=

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

Operands	Description	
function:		
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
	DISPLAY	Causes any buffered output to be written to the 4013.
ATTN=	NO, to disable the attention function.	
	YES, to enable the attention function.	
	This operand is required when function is SET.	

Examples:

```
ATTNOFF    TERMCTRL  SET,ATTN=NO
WRITEPTR   TERMCTRL  DISPLAY
```

4973 Printer

Syntax

```
label       TERMCTRL  function,LPI=

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

Operands	Description	
function:		
	SET	Sets the number of lines per inch. When SET is specified, the LPI operand is required.
	DISPLAY	Causes any buffered output to be written to the 4973.
LPI=	The number of lines, either 6 or 8, the 4973 is to print per inch. This operand is required when function is SET.	

Examples:

```
SETLPI6     TERMCTRL  SET,LPI=6
WRITEPTR    TERMCTRL  DISPLAY
```

4974 Printer

Syntax

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

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

Operands	Description
function:	
	DISPLAY — Causes any buffered output to be written to the 4974.
	PUTSTORE — Transfers control data from the processor to the 4974 wire image buffer. If PUTSTORE is specified, operands op1, op2, count, and TYPE are required.
	GETSTORE — Transfers control data from the 4974 wire image buffer to the processor. If GETSTORE is specified, operands op1, op2, count, and TYPE are required.
op1	The address in the processor from which or into which the information is to be transferred. Required when function is PUTSTORE or GETSTORE.
op2	The address in the 4974 wire image buffer to which or from which the information is to be transferred. Required when function is PUTSTORE or GETSTORE.
count	The number of bytes to be transferred. Required when function is PUTSTORE or GETSTORE.
TYPE=	The type of GETSTORE/PUTSTORE operation to be performed.
	1, to transfer data between the processor and the 4974 wire image buffer. If 1 is specified, function must be either PUTSTORE or GETSTORE.

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).

```
          TERMCTRL  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.

	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
	PUTSTORE	Transfers data from the processor to storage in the 4978. If specified, operands op1, op2, count, and TYPE= are required.
	GETSTORE	Transfers data from storage in the 4978 to the processor. If specified, operands op1, op2, count, and TYPE are required.

op1 — The address in the processor from which or into which the data is to be transferred.

op2 — The address in 4978 storage to which or from which data is to be transferred.

count — The number of bytes to be transferred.

ATTN= — NO, to disable the attention function.

YES, to enable the attention function.

This operand must be used in conjunction with the SET function.

TYPE= — 1, to indicate access to the character image buffer (a 2048-byte table, 8 bytes for each of the EBCDIC codes).

2, to indicate access to the control store (4096 bytes). The end condition (required when writing the control store) may be indicated by setting bit 0 on in the second operand. For example, to write the last 1024 bytes of the control store (#2 contains the control store address), code the following:

```
TERMCTRL PUTSTORE,BUFFER,(X'8000',#2),1024,TYPE=2
```

4, to indicate transfer of the field table from the device to the processor. If this option is specified, function must be GETSTORE. The input area must be defined with a BUFFER statement. At completion of the operation, the number of field addresses stored (addresses of unprotected fields) is placed in the control word at BUFFER-4.

5, to indicate transfer of the field table from the device to the processor. If this option is specified, function must be GETSTORE. A field table is transferred as for TYPE=4, but the addresses are those of the protected fields.

6, to indicate that the field table transferred contains only the addresses of modified fields. If this option is specified, function must be GETSTORE.

7, to indicate that the field table transferred contains address of the protected portions of modified fields. If this option is specified, function must be GETSTORE.

Examples:

```
TERMCTRL  BLANK                * BLANK SCREEN
    .
    .                          * MODIFY DISPLAY
    .
PRINTEXT  LINE=A,SPACES=B      * DEFINE CURSOR POSITION
TERMCTRL  DISPLAY * ENABLE DISPLAY

TERMCTRL  GETSTORE,BUFFER,0,2048,TYPE=1   * READ 4978
                                          * IMAGE STORE
```

4979 Display

Syntax

```
label       TERMCTRL   function,ATTN=

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

Operands	Description	
function:		
	BLANK	Inhibits display of the contents of the 4979 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.
	LOCK	Locks the keyboard.
	UNLOCK	Unlocks the keyboard.
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
ATTN=	NO, to disable the attention function.	
	YES, to enable the attention function.	

This operand must be used in conjunction with the SET function.

Examples:

```
TERMCTRL   BLANK               * BLANK SCREEN
   .
   .                           * MODIFY DISPLAY
   .
PRINTEXT   LINE=A,SPACES=B     * DEFINE CURSOR POSITION
TERMCTRL   DISPLAY             * ENABLE DISPLAY
```

ACCA Attached Devices

Syntax

```
label       TERMCTRL   function

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

Operands	Description	
function:		
	RING	Waits until the Ring Indicator (RI) is presented to the Series/1 from the modem. No timeout is provided.
	RINGT	Waits until the Ring Indicator (RI) is presented to the Series/1 from the modem. If no Ring Indicator (RI) occurs after 60 seconds, this instruction is terminated and an error condition is returned to the application program in the first word of the TCB.
	ENABLE	Activates Data Terminal Ready (DTR) if not jumpered on and waits for Data Set Ready (DSR) to be returned by the modem. No timeout is provided.

ENABLET	Activates Data Terminal Ready (DTR) if not jumpered on and waits for Data Set Ready (DSR) to be returned by the modem. If Data Set Ready (DSR) is not returned within 15 seconds, the instruction is terminated and an error condition is returned to the application program in the first word of the TCB.
ENABLEA	Activates Data Terminal Ready (DTR) if not jumpered on and waits for Data Set Ready (DSR) to be returned by the modem. When Data Set Ready (DSR) is returned, an answer tone is activated for three seconds. The modem must allow for the control of the answer tone.
ENABLEAT	Combines the functions of ENABLET and ENABLEA.
DISABLE	Disables Data Terminal Ready (DTR) if not jumpered on and waits for 15 seconds. This function is used to disconnect (hang up) the modem.

Examples:

```
TERMCTRL  RING                  * WAIT WITH NO TIMEOUT
TERMCTRL  DISABLE               * BREAK COMMUNICATION
```

ACCA Support Return Codes

Following each TERMCTRL instruction that is issued by an application program to an ACCA device, a return code is provided in the first word (taskname) of the TCB. The bits of the first word are interpreted as follows:

-1	Successful completion.
Bit	**Description**
0	Unused
1-8	ISB of last operation (I/O complete)
9-10	Unused
11	1 if a write or control operation (I/O complete)
12	Read operation (I/O complete)
13	Unused
14-15	Condition code +1 after I/O start (or) Condition code after I/O complete

Figure 15. Terminal I/O - ACCA Return Codes

Teletypewriter Equivalent Devices

Syntax

```
label        TERMCTRL   function,ATTN=

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

Operands	Description	
function:		
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
	DISPLAY	Causes any buffered output to be written to the teletypwriter.

ATTN= NO, to disable the attention function.

YES, to enable the attention function.

This operand must be used in conjunction with the SET function.

Examples:

```
SETATTN    TERMCTRL   SET,ATTN=NO
WRITEPTR   TERMCTRL   DISPLAY
```

Virtual Terminal

Syntax

```
label       TERMCTRL   function,code,ATTN=

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

Operands	Description	
function:		
	PF	Causes a simulated attention interrupt or program function key interrupt to be presented if the program is communicating with another program in the same processor (DEVICE=VIRT) or with a program in another processor (DEVICE=PROC).
		If the code is not specified or is zero, the keyboard task responds to the next READTEXT with '> ' and waits for an attention list code to be returned. If code has a non-zero value, x, the attention list code $PFx is automatically generated and the '> ' response does not occur.
		<u>note</u>: The 'code' may be a self-defining term or a variable containing the desired value.
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).

code — The attention or PF key value to be presented when using the PF function. This operand determines the attention or function key value.

ATTN= NO, to disable the attention function.

YES, to enable the attention function.

This operand must be used in conjunction with the SET function.

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 and if LENGTH is not coded, LENGTH=count.

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 messages will occur if LENGTH is exceeded. The maximum value is 254.

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
```

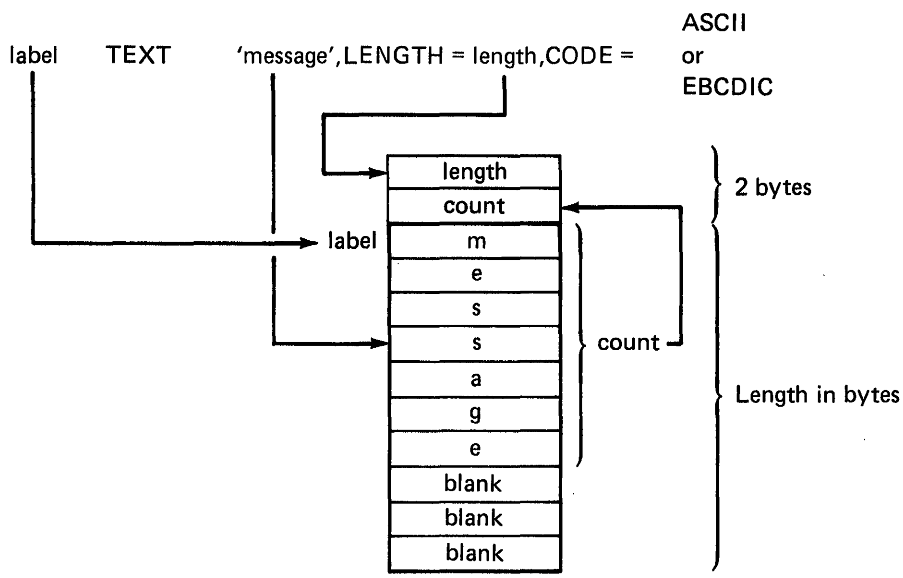


Figure 16. Text Statement

TITLE

Listing Control

The TITLE statement enables you to place a title at the top of each page of the assembly listings or at the top of individual pages. It is not supported by $EDXASM and will be treated as an EJECT instruction if encountered.

Syntax

```
blank        TITLE  message

Required:    message
Defaults:    none
```

Operands	Description
message	An alphameric character string up to 100 characters in length. This must be enclosed in apostrophes. A program may contain more than one TITLE statement. Each one causes following pages to begin with the new message. This heading is repeated on each new page following a TITLE statement until a new TITLE statement is encountered.

TP HOST COMMUNICATIONS (REFERENCE ONLY)

Telecommunications

TP OPENIN	TP OPENOUT	TP WRITE
TP CLOSE	TP SUBMIT	TP READ
TP SET	TP TIMEDATE	TP FETCH
TP RELEASE		

The Host Communications Facilities are described in the Communications and Terminal Applications Guide.

USER

Program Control

USER creates linkage to a user exit routine. This provides the user a means of programming (in Series/1 assembler language) a function which is not supported by the Event Driven Executive instruction set. The user exit routine must set the registers correctly to return control to the system at the end of the routine. Details of the conventions that must be followed are described below.

Syntax

```
 label       USER name,PARM=(parm1,...,parmn),
                  P=(name1,...,namen)

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

Operands	Description
name	The entry point name of the user-exit routine.
PARM=	A list of parameters that are to be passed to the user routine.
P=	A list of names to be attached to the PARM operands.

Upon entry to the user routine, register 1 points to the user's first parameter. If no parameters were passed to the user exit routine, register 1 will point to the address of the next statement following the USER instruction. Register 2 contains the address of the current tasks TCB. The user routine must preserve the contents of register 2 for eventual return to the supervisor. Your routine must also provide in register 1 the address of the next Event Driven language instruction to be executed when returning to the supervisor. If parameters are passed to the user routine, register 1 must be incremented within the user exit routine by double the number of parameters used before returning to the supervisor. If it is desired to return to an instruction other than the instruction following the USER statement, register 1 may be set to the address of the desired instruction. In all cases, the assembly language routine must exit by a branch to the label RETURN. Use of the USER routine requires that $LINK be used to include the RETURN object module, $$RETURN. The Autocall feature of $LINK may be used. See the Utilities, Operator Commands, Program Preparation, Messages and Codes for additional information. Figure 17 shows the control flow to a user exit routine.

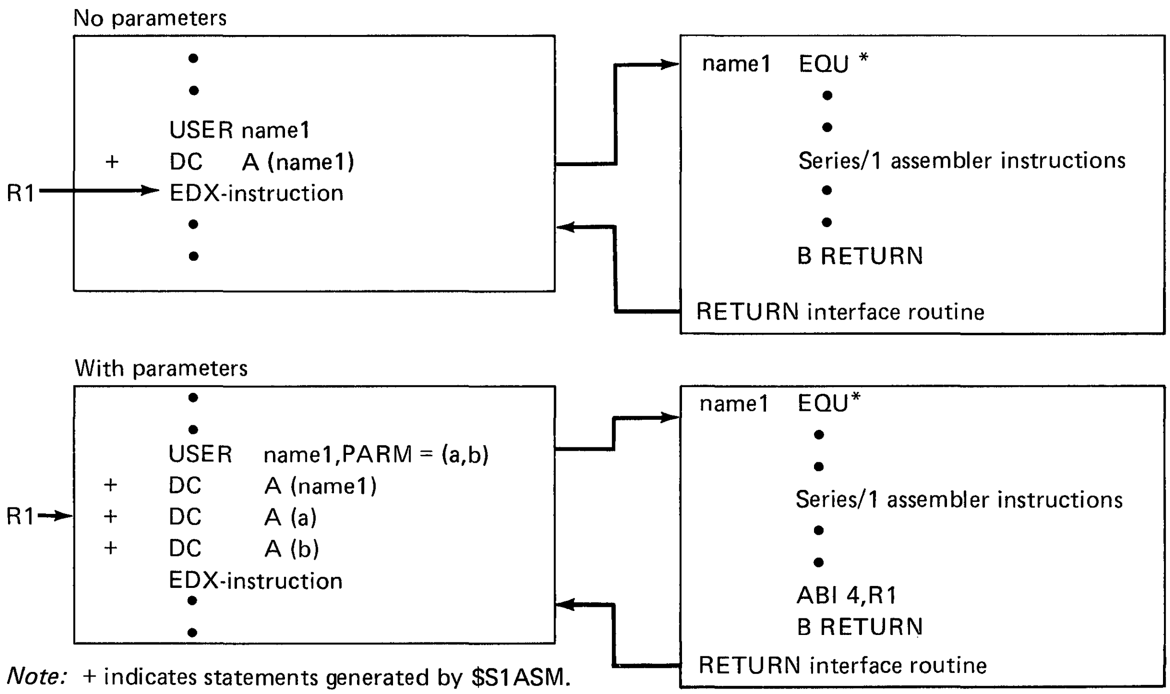


Figure 17. Calling A User Exit Routine and Returning

It will often be necessary for you to pass parameters to your routine. Parameters may be passed in the form of constants, which will be stored in the calling list, or as symbolic names (addresses) of the parameters. In the latter case, the address of the parameter is contained in the calling list. If the parameter is a constant, it may be addressed through register 1 which points to the first parameter on entry to the user routine. The instruction

```
MVW   (R1,0),R3
```

will load the parameter into Register 3.
The second parameter may be likewise loaded by

```
MVW   (R1,2),R3
```

The following instruction illustrates the method for acquiring a parameter (in this case, the second) whose address is passed in the calling sequence.

```
MVW   (R1,2)*,R3
```

The user exit routine is free to use all of the registers as long as registers 1 and 2 are set properly for return to the supervisor. The last instruction of the user routine must branch to RETURN which is an entry point in the interface module $$RETURN. This module must be link-edited with the user exit routine using the $LINK utility.

It may be useful in special cases to intermix Event Driven Language instructions and assembler instructions. This can be done by using the following coding convention:

```
            MOVE     A,B         STANDARD INSTRUCTION EXAMPLE
            ADD      A,10        ANOTHER INSTRUCTION

            USER     *+2         USER EXIT TO ASSEMBLER CODE
             .
             .
             .                   ASSEMBLER CODE
             .
      OK    BAL      RETURN,R1   SET REG 1 AND RETURN

            MOVE     B,A         NOW BACK INTO THE EVENT DRIVEN
            SUB      B,10        EXECUTIVE INSTRUCTION SET
```

In this example, the USER *+2 and BAL RETURN,R1 provide the linkage to assembler code and back to the supervisor, respectively. See "Example 10: User Exit Routine" on page 400 for an example of a user exit routine. The above coding technique can only be used with the Series/1 assembler or the host assembler. $EDXASM does not allow mixing Series/1 code with the Event Driven Language instructions.

If $EDXASM is used to assemble the source program, an EXTRN must be used to refer to the user exit routine. Then, $LINK must be used to link the module with the user exit routine module produced by the Series/1 or host assembler.

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.

WHERES

Task Control

WHERES is used to locate another program executing elsewhere in the system.

Syntax

```
label       WHERES  progname,address,KEY=,P1=,P2=,P3=

Required:   progname, address
Defaults:   none
Indexable:  none
```

Operands	Description
progname	The label of a 8 byte area containing the 1-8 character program name of the program to be located If less than 8 characters, the program name must be left-justified and padded with blanks.
address	The label of a word in which the loadpoint address of the located program will be returned if the program is found. This address is the first byte of the program and is also the beginning of the program header. If the program is not located, a -1 is stored at this location.
KEY=	The label of a word in which the address key of the partition containing the located program will be returned if the program is found. The address key is one less than the partition number.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions. P3 is the name of the KEY operand.

Each partition, beginning with partition number 1, is searched to determine if the named program is loaded there. Partitions are searched in ascending order. The return code placed in the task code word indicates the result of the search. If more than one copy exists, only the first one found is reported.

The WHERES function accomplishes communication among independently loaded programs. The address key value can be used as input to the cross-partition options of WAIT, POST, READ, WRITE, ATTACH, ENQ, DEQ, BSCREAD, BSCWRITE, and MOVE. The address can be used in conjunction with an application-defined convention to gain addressability to data or code routines within another program. One such technique is to obtain the contents of the $STORAGE word from the located program's header and use that to address data which the program has previously placed in its dynamic area. WHERES can also be used to determine if a particular program is already loaded, this can avoid the need to load another copy. Refer to the topic of "Cross-Partition Services" in the System Guide for additional information on the use of WHERES.

Return Codes

Code	Description
-1	Program found
0	Program not found

Example

```
          WHERES  PROG,ADDR,KEY=KEY
             .
             .
ADDR      DATA    F'0'
KEY       DATA    F'0'
PROG      DATA    C'PROGNAME'
```

After successful execution, ADDR contains the address of the program named PROGNAME and KEY contains the address key.

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 even numbered length from 18 to 32766 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 32766 and the value must be an even number. The value can either be expressed as 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 when WAIT=NO is coded.

ERROR= Use this operand to specify the first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If this operand is not specified, control will be returned to the next instruction after the WRITE and you must test for any errors. This operand must not be used if WAIT=NO is coded.

For tape, if END is not coded, an EOT will be treated as an error with an EOT return code. The ERROR path is taken for all return codes other than EOT or a -1. An attempt to write to a tape which has an unexpired date is also an error.

WAIT= If this operand is allowed to default, or if it is coded as WAIT=YES, the current task will be suspended until the operation is complete.

If the operand is coded as WAIT=NO, control will be returned after the operation is initiated and a subsequent WAIT DSx must be issued in order to determine when the operation is complete.

END and ERROR cannot be coded if WAIT=NO is coded. You must subsequently test the return code in the Event Control Block (ECB) named DSx or in the task code word (referred to by taskname). Two codes are of special significance. A -1 indicates a successful end of operation. A +10 indicates an End-of-Data-Set and may be of logical significance to the program rather than being an error. For programming purposes, any other return codes should be treated as errors.

Note: The return codes for disk/diskette and tape are listed later in this section.

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

WRITE normally assumes the buffer (loc operand) is in the same partition as the currently executing program. However, it is possible to use a buffer in another partition using the cross-partition capability of WRITE. See the System Guide section on Cross-Partition Services.

Disk/Tape Return Codes

Disk/tape I/O return codes are returned in two places:

- The first word of the DSCB (either DSn or DSCB name) 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 for disk and tape are shown in tables later in this section.

Following an error condition on tape, the read/write head position is immediately following the error record. The error retry has been attempted, but was unsuccessful. The count field, in the WRITE instruction, may or may not have been decremented to zero under this condition.

If detailed information concerning an error is desired, it may be obtained by printing all or part of the contents of the disk data blocks (DDBs) or tape data blocks (TDBs), located in the supervisor area of partition 1. This can be accomplished in either of two ways: (a) by using the $LOG utility (see System Guide for details of use), or (b) by using the following information. The starting address of the DDBs/TDBs may be obtained from the link-edit map of the supervisor. DDBs/TDBs can also be located by the field $DISKDDB in the communications vector table (CVT). Use the PROGEQU equate table to reference $DISKDDB, DDBEQU equate table for DDB, and the TDBEQU equate table for the TDB fields. The contents of the DDBs and the TDBs are described in the IBM Series/1 Event Driven Executive Internal Design, LY34-0168, under the headings of 'Disk Data Block', and 'DDB Equates'. 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/TDB in storage.

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 12 on page 249.

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.

Figure 18. 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

```
TASK1   PROGRAM   START1,DS=((OUTDATA,1025))
START1  WRITE     DS1,BUFF1,1,1000,ERROR=ERR
```

This example writes a single 1000 byte record from location BUFF1, to a tape data set named OUTDATA, on a standard labeled (SL) tape that has volume serial number 1025.

```
TASK2   PROGRAM   START2,DS=((OUTDATA,1025))
START2  WRITE     DS1,BUFF2,2,502,ERROR=ERR
```

This example writes two records to the tape data set. Each record is 502 bytes in length. Record one is located at BUFF2, record two is located at BUFF2 + 502 bytes.

WXTRN/EXTRN

Program Module Sectioning

Both of these statements identify symbols which are not defined within the program module containing the EXTRN/WXTRN statement. References to these symbols will be resolved when the program module is link edited with a program module containing an ENTRY definition for the subject symbol. If no such symbol is found during link-edit, the symbol is said to be unresolved and it is assigned the same address as the beginning of the program.

WXTRN symbols are resolved only by symbols that are contained in modules that are included by the INCLUDE statement in the link-edit process or by symbols found in modules that have been brought in by the AUTOCALL function. However, WXTRN itself does not trigger AUTOCALL processing.

Only symbols defined by EXTRN statements will be used as search arguments during the AUTOCALL processing function of $LINK. Any additional external symbols found in the module found by AUTOCALL will be used to resolve both EXTRN and WXTRN symbols. See the description of $LINK in Utilities, Operator Commands, Program Preparation, Messages and Codes for further information.

Syntax

```
   blank      WXTRN   One or more relocatable symbols
   blank      EXTRN   that are external to this
                      program, separated by commas

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

Operands	Description
	One or more external symbols which will be resolved by link editing to a program module which contains the same symbol defined by an ENTRY statement.

XYPLOT

Graphics

XYPLOT is used to draw a curve on the display connecting points specified by arrays of x and y values. Data values are scaled to screen addresses according to the plot control block, and points outside the plot area are placed on the nearest boundary.

Syntax

```
label        XYPLOT   x,y,pcb,n,P1=,P2=,P3=,P4=

Required:    x,y,pcb,n
Defaults:    none
Indexable:   none
```

Operands	Description
x	Address of array of x data values.
y	Address of array of y data values.
pcb	Label of 8-word Plot Control Block (see "PLOTGIN" on page 210 for a description of a Plot Control Block).
n	Address of location which contains number of points to be drawn.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

Example: See "Example 12: Graphics Instructions Programming Example" on page 408 for an example of coding XYPLOT.

YTPLOT

Graphics

YTPLOT is used to draw a curve on the display connecting points equally spaced horizontally and having heights specified by an array of y values. Data values are scaled to screen addresses according to the plot control block, and points outside the range are placed on the boundary of the plot area.

Syntax

```
label       YTPLOT   y,x1,pcb,n,inc,P1=,P2=,P3=,P4=,P5=

Required:   y,x1,pcb,n,inc
Defaults:   none
Indexable:  none
```

Operands	Description
y	Address of array of y data values.
x1	Address of x data value associated with first point.
pcb	Label of 8-word Plot Control Block (see "PLOTGIN" on page 210 for a description of a Plot Control Block).
n	Address of location which contains number of points to be drawn.
inc	Explicit value of increment of x data value between points. Inc must not be zero and must be an explicit value.
Px=	Parameter naming operands. See "Use of The Parameter Naming Operands (Px=)" on page 8 for further descriptions.

This chapter describes the requests that make the Indexed Access Method available to the user: PROCESS, LOAD, GET, GETSEQ, PUT, PUTUP, PUTDE, RELEASE, DELETE, ENDSEQ, EXTRACT, and DISCONN. Included for each request is a description of the purpose of the request, the detailed coding syntax, a description of each parameter, and all of the return codes associated with use of these requests. The Indexed Access Method Licensed Program must be installed to use any of these services.

The information in this chapter is intended for use as a reference when coding. For a complete description of the Indexed Access Method refer to System Guide.

All Indexed Access Method services are requested by use of the CALL instruction. Call parameters can have the following forms:

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

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

For additional information refer to "CALL" on page 68.

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

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

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

Programs which call the Indexed Access Method must be processed by $LINK to include the subroutine module IAM. IAMEQU has an EXTRN statement for IAM. Refer to the chapter on "$LINK" in Utilities, Operator Commands, Program Preparation, Messages and Codes for information on how to perform the link edit process.

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

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

The return codes associated with each request are included with the description of the request. Parameters parm3, parm4, and parm5 are set to zero by the Indexed Access Method before returning. These parameters must be reinitialized before executing the CALL instruction again.

"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.

DELETE

Indexed Access Method

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

Syntax

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


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

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS.
(key)	The label of your key area containing the 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.

Return Codes

Code	Condition
-1	Successful
-85	Record not found
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
80	Write error - FCB
100	Read error
101	Write error

Example

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

DISCONN

Indexed Access Method

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

Syntax

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


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

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

Return Codes

Code	Condition
-1	Successful
7	Link module in use
8	Unable to load $IAM
12	Data set shut down
13	Module not included in load module
22	Invalid IACB address
100	Read error
101	Write error
110	Write error, data set closed

ENDSEQ

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

Syntax

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

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

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

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)


Required:   all
Defaults:   size=full FCB
Indexable:  none
```

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS or LOAD.
(buff)	The label of the user area into which the File Control Block (FCB) is returned. The area must be large enough to contain the requested portion of the FCB. Use the COPY statement to include FCBEQU in your program so that the FCB fields can be referenced by symbolic names.
(size)	The number of bytes of the area into which the FCB is to be copied. A full FCB requires 256 bytes. The symbol FCBSIZE in the FCBEQU equate table represents the actual size of the data in the FCB and can be used as this parameter.

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

GET

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.

Syntax

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

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

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

(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
22	Invalid IACB address
100	Read error
101	Write error

GETSEQ

Indexed Access Method

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

The first GETSEQ of a sequence is performed like a GET; the first record in the data set that satisfies the key conditions is the one that is retrieved. If key is zero, the first record in the data set is retrieved. Subsequent requests in the sequence locate the next sequential record in the data set and the key parameter is ignored if specified. The sequence is terminated by an end-of-data condition, by an ENDSEQ, by a DISCONN, or by an error. During the sequence, direct-access requests are invalid. Retrieval for update can be specified if the requested record is intended for possible modification or deletion. If update is used the record is locked and remains unavailable to any other requests until the update is completed by a PUTUP, PUTDE or RELEASE. The record is also released by ending the sequence with an ENDSEQ or by ending processing with a DISCONN or by an occur.

During an update, the user must 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.

Syntax

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

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

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

(buff) The label of the user area into which the requested record is placed.

(key) The label of the user key area containing the key identifying the record to be retrieved and preceded by the lengths of the key and area. If the first record of the data set is to be retrieved, this field as specified should be 0. The key field, if specified, has the standard TEXT format and may be declared using the TEXT statement. 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

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
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.

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

Offset	Field
0	SYSRTCD
2	ERREXIT
4	(0) reserved

Where:

SYSRTCD A 1-word variable in which the system return code is placed if a system function requested under this LOAD by the Indexed Access Method terminates with a positive return code.

ERREXIT The user's error exit routine address. If this address is zero, the error exit will not be taken. Note that error exits handle only positive returns.

RESERVED Must be 0 for LOAD requests.

(mode) Specifies shared or exclusive use of the data set.

SHARE Allows shared read/write access by PROCESS requests.

EXCLUSV You are allowed access to the data set only if there are no outstanding PROCESS or LOAD requests. No other user can access the data set while exclusive use is granted to another.

Return Codes

Code	Condition
-1	Successful
-57	Data set has been loaded
7	Link module in use
8	Unable to load $IAM
12	Data set shut down
13	Module not included in load module
23	Insufficient IACBs
50	File opened exclusive
51	Opened in load mode
52	File in use, cannot open exclusive
54	Invalid blocksize
55	Insufficient FCBs
56	Read error - FCB

Example

```
         CALL   IAM,(LOAD),IACB,(DS3),(OPEN),(EXCLUSV)
           .
           .
           .
   IACB  DATA   F'0'
   OPEN  DATA   F'0'         return codes
         DATA   A'ERROR'     error exit
         DATA   F'0'         not used
```

PROCESS

Indexed Access Method

The PROCESS 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 PROCESS to this data set.

PROCESS opens the data set for retrievals, updates, insertions, and deletions. Multiple users can PROCESS the same data set. However, only one user at a time can use the LOAD function for a given data set.

If ERREXIT is specified, the error exit routine is executed whenever any Indexed Access Method request under this PROCESS terminates with a positive return code. If EODEXIT is specified, the end-of-data exit routine is executed whenever a GETSEQ associated with PROCESS attempts to access a record after the last record in the data set.

Syntax

```
label       CALL  IAM,(PROCESS),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.

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

Offset	Field
0	SYSRTCD
2	ERREXIT
4	EODEXIT

Where:

SYSRTCD A 1-word variable in which the system return code is placed if a system function requested under this PROCESS by the Indexed Access Method terminates with a positive return code.

ERREXIT Your error exit routine address. If this address is 0, the error exit will not be used. Note that error exits handle only positive return codes.

EODEXIT Your end-of-data exit routine address. If this address is 0, the end-of-data exit will not be used.

(mode) Specifies shared or exclusive access to the data set.

SHARE Allows shared read/write access by multiple PROCESS or LOAD requests.

EXCLUSV The user is allowed access to the data set only if there are no outstanding PROCESS or LOAD requests. No other user can access the data set while exclusive use is granted to another.

Return Codes

Code	Condition
-1	Successful
-57	Data set has been loaded
7	Link module in use
8	Unable to load $IAM
12	Data set shut down
13	Module not included in load module
23	Insufficient IACBs
50	File opened exclusive
51	Opened in load mode
52	File in use, cannot open exclusive
54	Invalid blocksize
55	Insufficient FCBs
56	Read error - FCB

Example

```
        CALL   IAM,(PROCESS),IACB,(DS1),(OPENTAB),(SHARE)
         .
         .
OPENTAB DATA   F'0'           return codes
        DATA   A(ERROR)       address of error exit
        DATA   A(END)         address of EOD exit
IACB    DATA   F'0'
         .
```

PUT

Indexed Access Method

The PUT request processes the record that is in your buffer (buff) according to the way the data set was opened (LOAD or PROCESS).

If the current open is for LOAD, the record must have a higher key than the highest key already in the data set and only base records are used (refer to the System Guide for information on LOAD mode). If the current open is for PROCESS, the record may have any key and is placed in either a base or a free record slot.

Syntax

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

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

Operands	Description
iacb	The label of a word containing the IACB address returned by PROCESS or LOAD.
(buff)	The label of the user area containing the record to be added to the data set.

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
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
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.

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
85	Key was modified by user
100	Read error
101	Write error

RELEASE

The RELEASE request frees a record that has been locked by a GET or GETSEQ for update. A record lock is normally released by a PUTUP or PUTDE. The RELEASE request is useful for freeing the locked record when the update need not be completed. RELEASE is valid only when a record is locked for update.

Syntax

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


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

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

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

The services of the Multiple Terminal Manager are requested using the instruction "CALL" on page 68. This section describes each of the functions and the coding syntax of the CALL, the parameters and the return codes.

The use, purpose, and messages for the Multiple Terminal Manager functions are described further in the Communications and Terminal Applications Guide.

The general format of a Multiple Terminal Manager request is:

```
CALL  routine,(parm1),(parm2).....
```

All parameters are enclosed in parentheses and are the addresses of variables in the requesting program.

ACTION

Multiple Terminal Manager

ACTION begins the prompt-response terminal I/O cycle. For IBM 4978/4979/3101 displays the parameter list is ignored (if specified). The input buffer is written protected to the screen if a CALL SETPAN or CALL CHGPAN command was executed previously during this execution. The output buffer is scatter written into the unprotected fields on the screen. If no SETPAN or CHGPAN precedes the ACTION, only the output buffer is written. The terminal then waits for operator input and reenters the current program (with operator input in the input buffer) at the next sequential instruction after CALL ACTION.

For asynchronous terminals, ACTION does the following:

1. Writes the specified buffer contents to the terminal (performs the Multiple Terminal Manager WRITE function).
2. Waits for the operator to respond
3. Reenters the current program at the instruction following the CALL ACTION.

Syntax

```
label        CALL ACTION,(buffer),(length),(crlf)


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

Operands	Description
(buffer)	A buffer of EBCDIC text of any length.
(length)	The number of characters in the buffer.
(crlf)	A binary value of 1 specifies that the terminal is to be issued a CR and LF after the message is sent. Any other value results in no CRLF being sent.

BEEP

Multiple Terminal Manager

CALL BEEP causes the audible alarm (optional feature) to be sounded at the current terminal following the next output cycle. The IBM 4979 terminal has no audible alarm and ignores this request. The current display and cursor position for 4978, 4979 and 3101 are not affected. When executed for an asynchronous terminal, this request causes the next output line to be followed by a bell character.

Syntax

```
label       CALL BEEP


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

Operands	Description
none	none

CDATA

Although the terminal environment block can be accessed directly because its address is a user program parameter, you may find it more convenient with your program to use the CDATA function, to determine the attributes of the calling terminal. The CDATA subroutine returns data concerning the terminal which is currently executing the program.

Syntax

```
label       CALL CDATA,(type),(userid),(userclass),
                      (termname),(buffsize)


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

Operands	Description
(type)	A word where: 0 = Terminal is an IBM 4978, 4979, or 3101 2 = Terminal is an ASR 33/35 or equivalent
(userid)	The 16-bit value returned by the SIGNON program when the current terminal signed on. If the current terminal does not use SIGNON, this value is set to zero.
(userclass)	The 16-bit value returned by the SIGNON program when the current terminal signed on. If the current terminal does not use SIGNON, this value is set to zero.
(termname)	The label of a field containing the 8-byte name (right padded with blanks, if necessary) of the current terminal.

(buffsize) The length of the terminal's input buffer. For asynchronous terminals this is 150 bytes. For IBM 4978, 4979, or 3101 terminals, this is the number of unprotected blanks in the last screen panel which was set using SETPAN or CHGPAN.

CHGPAN

Multiple Terminal Manager

After a CALL SETPAN, the protected characters of the screen panel specified have been placed in the input buffer. You can add data to the image prior to the next output cycle, and the data is displayed as protected data. If you do this, you must also CALL CHGPAN to inform the Multiple Terminal Manager that it needs to recompute the location of the first unprotected character position in the current panel and the count of unprotected characters. CHGPAN also sets the SETPAN indicator, allowing applications to dynamically develop protected screen panels.

Syntax

```
label       CALL CHGPAN


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

Operands	Description
none	None

CYCLE

Multiple Terminal Manager

When CALL CYCLE executes, the program may be swapped out as all other terminals are given an opportunity to process inputs. The program and the output buffer are preserved but the contents of the input buffer are lost (set to blanks). If a SETPAN or CHGPAN has been executed, the screen in the input buffer is displayed protected at this time to free up the input buffer.

After all other terminals have processed their inputs, the program is swapped into the program area and control is returned to the instruction after the CALL CYCLE.

Syntax

```
label       CALL CYCLE


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

Operands	Description
none	none

FAN

Multiple Terminal Manager

The FAN function is a no-operation in the EDX environment. It is provided only for compatibility with other implementations.

Syntax

```
label        CALL FAN


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

Operands	Description
none	none

FILEIO

Multiple Terminal Manager

When executing programs under the Multiple Terminal Manager, all requests for disk/diskette I/O are by means of a call to the FILEIO routine. FILEIO provides the following functions:

- Automatic open of the requested data set.
- Direct access support where records are accessed by a relative record number (RRN).
- Support for Indexed Access Method files, providing a high-level language interface to most Indexed Access Method services.
- Data integrity, using automatic close of terminal manager shutdown and automatic write back of data buffers.

FILEIO automatically controls the opened/closed status of a data set. Thus data set names must not be coded on the PROGRAM statement of programs to be executed using the Multiple Terminal Manager. If the data set is not open when a request is made, the data set is opened. Because many terminals can require many data sets, some common and some unique, you may find that there is no storage available to open a requested data set. In order to avoid this situation, a limit is set for the number of open data sets. Multiple Terminal Manager is distributed with space allocated for 14 open data sets. When this limit is reached, the least recently accessed data set is closed, and the space it required is reused. A data set is not available for automatic close if it has an update pending. The user can adjust the maximum number of open data sets by changing the file table in the Multiple Terminal Manager source module CDMCOMMN.

FILEIO provides the facility to access previously created files using the CALL interface described earlier. These files must have been previously defined and, if indexed, they must have been previously loaded.

<u>Syntax</u>

FILEIO

```
label       CALL FILEIO,(fca),(buffer),(code)

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

Operands	Description
(fca)	File Control Area (FCA) - The label of a table with the parameters describing the requested operations. Some fields are defined differently depending on the request type specified. The format of the FCA is shown below:

0	Request Type	A 4-byte EBCDIC request Valid request types are shown below
4	Data Set Name	An 8-byte EBCDIC data set name, left justified and padded with blanks
12	Key Relation	A 2-byte EBCDIC key rela-ion Operator, GT, GE, EQ (indexed files only)
12	Number of Records	A word value for the number of 256 byte records to be read or written (direct file requests only)
14	Key Length	A word specifying the lengthof the key to be used for retrieval. If the length specified is less than the actual key length, the first n bytes of the key are used (indexed files only)

16	Key Location	The address of the key to be used (indexed requests only). For a COBOL program, this should be zero.
or		
16	EOD Record Number	The system-maintained logical EOD record number passed back to the application after each direct file READ or WRITE (direct file requests only).
18	Reserved	Must be zero
20	Relative Record Number (RRN)	A word value for the RRN. The first record is record number 1 (direct file requests only)
22	Volume Name	A 6-byte EBCDIC volume name left justified and padded with blanks
28	Search Key	Used only by COBOLpro- grams to specify the key for indexed requests

(buffer) The name of the user program I/O buffer. This buffer contains the record to be written or receives the record read.

(code) The name of the 2-byte field to contain the return code passed back by FILEIO. This can be a FILEIO return code, a system error code or it can be passed from the Indexed Access Method.

Indexed File Request Types

RELS Release from sequential processing mode (ENDSEQ)

RELR Release a record held for update (RELEASE)

PUTU Put operation, update mode (PUTUP)

PUTD Put operation, delete mode (PUTDE)

PUTN Put operation, new mode, adds a record to the file (PUT)

GETD Get operation, direct read (GET)

GETS Get operation, sequential read (GETSEQ)

IDEL Delete operation (DELETE)

ICLS Close an Indexed data set (DISCONN)

GTRU Direct get, update mode (GET)

GTSU Sequential get, update mode (GETSEQ)

Indexed file requests cause invocation of the Indexed Access Method function shown in parentheses. Files are accessed in the PROCESS mode and are shared. See "Chapter 4. Indexed Access Method" on page 327 for more information.

Direct File Request Types

READ Read the record defined by the RRN field of the FCA into the user-provided buffer

WRIT Write the record defined by the RRN field of the FCA into the user-provided buffer

SEOD Set the system maintained EOD pointer to the record number provided in the RRN field of the FCA

FILEIO Return Codes

Code	Description
-1	Successful operation
201	Data set not found
202	Volume not found
203	No file table entries are available; all have updates outstanding.
204	I/O error reading volume directory
205	I/O error writing volume directory
206	Invalid function request
207	Invalid key operator
208	SEOD record number greater than data set length

Other return codes not shown above are returned by the Indexed Access Method or by the system data management support.

For further information on CALL FILEIO see the Communications and Terminal Applications Guide

FTAB

Multiple Terminal Manager

The FTAB function sets up a table which describes the unprotected (input) fields placed in the input buffer following a SETPAN or CHGPAN operation. This description is useful for such things as positioning the cursor to a specific field or to a precise location within a field.

The FTAB code must be included in the application link-edit step in order to be available to the application program. Refer to the Utilities, Operator Commands, Program Preparation, Messages and Codes for details on link-editing.

Syntax

```
label       CALL   FTAB,(table),(size),(return code)

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

Operands	Description
table	The table operand is made up of a sequence of three-word entries. Each three-word entry describes an unprotected field of the screen image in the input buffer in the order: row, column, and length. The sequence begins at the location of the variable named in the table operand and is repeated for each successive field of the screen. Following is an example of the table format:

```
TABLE       row          (word one of the first field)
            column       (word two of the first field)
            length       (word three of the first field)
TABLE+6     row          (second field)
            column
            length
TABLE+12    row          (third field)
            column
            length
  .           .
  .           .
  n
where n is equal to the value of the size operand
```

size — This operand is one word long and contains the number of entries in the table. This decimal value can be in the range 1 to 32767.

Note: Unused fields in the FTAB table are always set to zero.

return code

This operand is the name of a one-word field reserved for a return code that represents the completion status of the FTAB function.

Return Codes

Code	Description
-2	FTAB code not link-edited with application
-1	successful return
1	no data fields found
2	data table truncated

LINK

Multiple Terminal Manager

A CALL to LINK causes the named Multiple Terminal Manager program to be loaded and executed (replacing the current program).

If a SETPAN or CHGPAN precedes the LINK, the contents of the input buffer will be displayed for 4978, 4979, or 3101 terminals and the buffer freed. The output buffer is passed unchanged to the next program.

The program being linked to receives the standard parameter list for application programs (input buffer, output buffer, etc.).

Syntax

```
label        CALL LINK,(pgmname)

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

Operands	Description
pgmname	The name of the variable containing the 8-byte program name (right padded with blanks, if necessary).

If the program name is invalid or cannot be found, this module returns to the caller; therefore, any return to the user from CALL LINK is an error condition.

Example

```
PROG      CALL    LINK,(PROG)
          GOTO    ERROR
          .
          .
PROG      DATA    C'PROGNAME'
```

LINKON

A CALL to LINKON provides the same function as CALL LINK, except that a full output cycle is taken and the terminal manager waits for an operator response. The named program is then entered at its entry point with the input buffer containing the unprotected characters from the screen or all the characters entered from the asynchronous terminal.

Syntax

```
label       CALL LINKON,(pgmname)

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

Operands	Description
(pgmname)	The name of a variable containing the 8-byte program name (right padded with blanks, if necessary).

If the program name is invalid or cannot be found, this module returns to the caller; therefore, any return to the user from CALL LINKON is an error condition.

Example

```
            CALL    LINKON,(PROG)
            GOTO    ERROR
            .
            .
PROG        DATA    CL8'PROG'
```

MENU

Multiple Terminal Manager

CALL MENU immediately aborts the current dialogue and causes the terminal's menu screen (or request for program name message) to be displayed.

The operator can cause this at any time by pressing PF3 on an IBM 4979/4978/3101, or by typing OUT on an asynchronous terminal while in a dialogue.

Syntax

```
label       CALL MENU


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

Operands	Description
none	none

SETCUR

CALL SETCUR specifies the position at which the cursor is to be displayed for the next output cycle. The cursor position is expressed as a pair of row and column coordinates on the screen.

Each screen panel specifies a cursor position to be used while the screen is active (until the next SETPAN or CHGPAN). This function permits you to override the cursor position for the next output.

Syntax

```
label        CALL SETCUR,(row),(column)

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

Operands	Description
(row)	The label of a word containing the row number at which the cursor is to be set. Allowable row numbers are 0-23; row 0 is the top line of the screen.
(column)	The label of a word containing the column number at which the cursor is to be set. Allowable column numbers are 0-79; column 79 is the rightmost position of a row.

Example:

Set cursor position to lower righthand corner of a 4978 or 3101 display.

```
          CALL    SETCUR,(ROW),(COLUMN)

ROW       DATA    F'23'            BOTTOM LINE
COLUMN    DATA    F'79'            RIGHTMOST CHARACTER
```

SETPAN

The SETPAN routine causes the specified screen format to be read into the input buffer (replacing the last operator input) and sets a switch to cause the screen format to be written to the screen during the next output cycle. Any nulls (X'00') in the screen image will be written unprotected. All other characters will be written protected. In addition to placing the 1920 byte screen panel into the input buffer, any unprotected defaults that were specified when the screen was built are moved, concatenated, into the output buffer. The cursor position for the next display after SETPAN will be set to the first unprotected character position. Before executing a CALL SETPAN, be sure to save any needed information which is in the buffers as it will be overlaid by the panel definition.

Syntax

```
label        CALL SETPAN,(dsname),(code)


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

Operands	Description
(dsname)	The name of a variable containing the 8 byte data set name of the desired screen format in the SCRNS volume.
(code)	The label of a word in which SETPAN will place a return code.

The SETPAN return codes are:

Code	Description
-501	Screen data set not found.
-500	This terminal is not an IBM 4978/4979/3101; no action has been taken.
-1	Successful, new panel in buffer
1	Warning, the data set does not contain a valid screen image. The input buffer has been set to unprotected blanks (X'00') and the cursor position set to 0.
2	Warning, too many unprotected default characters in the screen definition. The number of default characters that will be displayed has been truncated. This return code will be received if there are no default unprotected characters in the screen. The $IMAGE utility initially assigns 1920 unprotected characters to a screen. This number is unchanged if the data (non-protected) was not modified using the edit mode of the $IMAGE utility.
Other	Return code from disk READ.

Example

```
         CALL    SETPAN,(SCREEN01),(RC)
         .
         .
         .
SCREEN01 DATA    C'SCREEN01'
RC       DATA    F'0'
```

WRITE

Multiple Terminal Manager

Note: The EDL task control WRITE is located under "WRITE" on page 317.

The Multiple Terminal Manager provides CALL WRITE to write output messages to asynchronous terminals without allowing operator response. It writes the specified buffer contents to the current terminal. While writing, other terminals are permitted to operate. When I/O is complete, the current user program is reloaded and reentered at the next sequential instruction after CALL WRITE.

No operator entry is permitted (see ACTION if operator entry is required). Printers and 4978/4979 displays are not supported by CALL WRITE.

Syntax

```
label       CALL WRITE,(buffer),(length),(crlf)


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

Operands	Description
(buffer)	The label of a buffer of EBCDIC text of any length.
(length)	The label of a word containing the number of characters in the buffer.
(crlf)	The label of a word specifying the CRLF option. A binary value of 1 in a word specifies that the terminal is to be issued a CR and LF after the message is sent. Any other value results in no CRLF being sent.

If no CRLF is specified (crlf word is not 1), trailing blanks in the buffer are transmitted to permit you to position the terminal for the next message or operator response.

WRITE

The Multiple Terminal Manager does not keep track of current terminal cursor or carriage position. No CRLF is inserted if, due to messages without CRLF, or a buffer size larger than the terminal line length, the right margin is reached.

If executed when using an IBM 4978/4979, control returns immediately to the caller.

Upon completion, the contents of the buffer are unchanged.

CHAPTER 6. PROGRAMMING EXAMPLES

In this chapter several examples are presented to demonstrate the use of the Event Driven Language instructions for typical applications.

It should be noted that most of the examples shown here are not complete programs in that they do not contain PROGRAM, IODEF, ENDPROG, and END statements.

Following is a list of the examples that are included, along with the title of each example.

Example 1 -- Read and print date

Example 2 -- Analog input

Example 3 -- Analog input with buffering to disk

Example 4 -- Digital input and averaging

Example 5 -- Index register usage

Example 6 -- Use of MOVEA

Example 7 -- A two task program with ATTNLIST

Example 8 -- Program loading functions

Example 9 -- Floating point, WAIT/POST, GETEDIT/PUTEDIT

Example 10 -- User exit routine

Example 11 -- I/O Level control program

Example 12 -- Graphics example

Example 13 -- Format and display trace data

Example 14 -- Use of Indexed Access Method

Example 15 -- Write data to tape data set

Example 16 -- Processing standard labels using BLP

Example 17 -- Write a data set to a SL tape then READ it

Example 18 -- Initialize and WRITE a NL tape

Example 19 -- READ the third file on tape

EXAMPLE 1: READ AND PRINT DATE

Read in and print the date on a terminal.

```
*   ENQUEUE FOR THE TERMINAL
    START     ENQT
              PRINTEXT  '@EXAMPLE 1 - ENTER THE DATE@'
              GETVALUE  DAY,' DAY ? '
              GETVALUE  MONTH,' MONTH ? '
              GETVALUE  YEAR,' YEAR ? '
              PRINTEXT  ' THE DATE ',SKIP=5
              PRINTNUM  DAY,3     PRINT DAY, MONTH & YEAR
              PRINTEXT  SKIP=1    SKIP TO NEW LINE
*   DEQUEUE TERMINAL
              DEQT
*     . . . CONTINUE PROGRAM

DAY           DATA      F'0'
MONTH         DATA      F'0'
YEAR          DATA      F'0'
```

The program enqueues for the terminal in order to provide uninterrupted use while keying in the date and printing it back. An introductory message is typed, preceded and followed by carriage returns, followed by three input requests using the GETVALUE instruction. Five lines are skipped, and the date message is printed. Since the DAY, MONTH, and YEAR are stored in adjacent locations, only one PRINTNUM statement is needed to print all three numbers. At the end of the program, the terminal is dequeued to allow access by other users.

In this example, the program may be simplified by using one GETVALUE instruction to read all three values. The output from this program is illustrated in the following example.

```
EXAMPLE 1 - ENTER THE DATE
  DAY ?  30
  MONTH ? 7
  YEAR ?  79

  THE DATE:     30   7   79
```

EXAMPLE 2: ANALOG INPUT

This program takes 256 samples from analog input address AI1 at a sampling rate of 10 points/second. Set the run light on in the lab at the start of the run and turn it off at the end. The run light is connected to bit 3 of group DO2.

```
TKNAME     PROGRAM    START
           IODEF      DO2,TYPE=GROUP,ADDRESS=87
           IODEF      AI1,ADDRESS=83
START      SBIO       DO2,1,BITS=(3,1) TURN ON RUN LIGHT
*
           DO         256,TIMES        SET UP FOR 256 PTS
           STIMER     100              SET TIMER FOR 100 MS
           SBIO       AI1,BUFR,INDEX   READ AI1 WITH
*    AUTOMATIC INDEXING INTO THE BUFFER 'BUFR'
*    AND THEN WAIT FOR THE TIMER TO EXPIRE
           WAIT       TIMER
           ENDDO                        END OF LOOP
*
           SBIO       DO2,0,BITS=(3,1) TURN OFF RUN LIGHT
*
*    . . . CONTINUE PROGRAM
*
BUFR       BUFFER     256               256 WORD BUFFER
```

The program begins by writing a 1 into bit 3 of digital output group DO2. A loop is initialized for 256 cycles using the DO command. At this point, a software timer is set up for 100 milliseconds to provide sampling at 10 points/second. The analog data is read into BUFR using the SBIO instruction with automatic indexing. After the data is read, the program waits for the timer to expire before returning for the next sample. When all the data is collected, the run light is turned off by writing a 0 into bit 3 of DO2.

EXAMPLE 3: ANALOG INPUT WITH BUFFERING TO DISK

This program takes analog data readings at equal time intervals. The number of data points and the time interval in milliseconds are read in from the operator's terminal. The program will allow from 10 to 10,000 data points to be taken at time intervals between 10 milliseconds and 10 seconds (10,000 msec). The data collection is initiated by a process interrupt start signal. The program is aborted by using the keyboard function 'AB'. Also, a second keyboard function, 'NP', is used to print a status switch. The switch will be equal to zero if the start signal has not been received or equal to the number of data points to be read if the start signal has been received and data collection has begun.

```
*         TITLE 'SAMPLE ANALOG DATA ACQUISITION PROGRAM'
*
*
READATA   PROGRAM BEGIN,DS=??
          ATTNLIST (AB,ABORT,NP,SWPRNT)
*
*    ABORT THE EXPERIMENT
*
ABORT     MOVE  SWITCH,1
          ENDATTN
*
*    PRINT OUT EXPERIMENT SWITCH
*
SWPRNT    PRINTEXT TXT10
          PRINTNUM SWITCH
          PRINTEXT SKIP=1
          ENDATTN
*
          IODEF    AI1,ADDRESS=91,POINT=0
          IODEF    PI1,ADDRESS=94,BIT=15
*
*    EXPERIMENT INITIALIZATION
*
BEGIN     PRINTEXT TXT1
          GETVALUE RUNUM,TXT2 REQUEST RUN IDENTIFIER
GETINT    GETVALUE INTVL,TXT3  REQUEST TIME INTERVAL
          IF  (INTVL,LT,10),OR,(INTVL,GT,10000),GOTO,GETINT
GETPTS    GETVALUE NPTS,TXT4   REQUEST NO. OF POINTS
          IF  (NPTS,LT,10),OR,(NPTS,GT,10000),GOTO,GETPTS
*
          WRITE DS1,RUNUM      RUN PARAMETERS IN 1ST SECTOR
          RESET SWITCH
          PRINTEXT TXT9        PRINT READY MESSAGE
          WAIT  PI1,RESET      WAIT FOR START SIGNAL
          MOVE  SWITCH,NPTS    SET SWITCH TO NPTS
```

```
*     THIS IS THE DATA ACQUISITION PORTION OF THE PROGRAM
*
         DO     NPTS           LOOP COUNT SET ABOVE
         STIMER INTVL       TIME INTERVAL SET ABOVE
         SBIO   AI1,BUFFER,INDEX  READ A DATA POINT
         IF     (BUFINDEX,EQ,128),GOTO,ATTACH  1ST BUFFER
                                                    FULL?
         IF     (BUFINDEX,NE,256),GOTO,TWAIT   ..NO, IS 2ND
                                                    FULL?
         MOVE   BUFINDEX,0      ..YES, RESET BUFFER INDEX
         ADD    POINTCNT,256    INCREMENT DATA COUNTER
*
ATTACH   IF     (DISK,NE,-1),GOTO,STOP    IS DISK TASK
                                            ATTACHED?
*  START DISK OUTPUT TASK
         ATTACH DISKTASK
*
TWAIT    WAIT   TIMER          WAIT FOR END OF TIME INTERVAL
         IF     (SWITCH,EQ,1),GOTO,STOP  TEST FOR 'ABORT'
ENDLOOP  ENDDO
*
         IF  (BUFINDEX,EQ,0),OR,(BUFINDEX,EQ,128),GOTO,STOP
         WAIT   DS1          ..YES, WAIT FOR DISK WRITE
         ADD    POINTCNT,BUFINDEX  UPDATE DATA COUNTER
         ATTACH DISKTASK       START LAST DISK OUTPUT
*
STOP     WAIT   DS1           WAIT FOR LAST OUTPUT OPERATION
         ENQT                  GET CONTROL OF TERMINAL
         PRINTEXT TXT6         PRINT TERMINATING MESSAGE
         PRINTNUM POINTCNT
         PRINTEXT TXT7
         DEQT                  RELEASE TERMINAL
         PROGSTOP
*
*
*              THIS IS THE DATA RECORDING TASK
*        IT IS ATTACHED BY THE DATA ACQUISITION TASK EACH
*        TIME THAT 128 WORDS OF DATA HAVE BEEN READ IN.
*        ONE PORTION OF THE BUFFER WILL BE TRANSFERRED TO
*        DISK WHILE DATA IS SIMULTANEOUSLY BEING READ INTO
*        THE OTHER PORTION OF THE BUFFER.
*
*        THIS TASK RUNS ON LEVEL 3 AT A LOWER PRIORITY THAN
*        THE DATA ACQUISITION TASK IN ORDER TO MAXIMIZE
*        TIMING ACCURACY.
*
DISKTASK TASK   DISK1,300,EVENT=DISK
DISK1    WRITE DS1,BUFFER1,ERROR=DISKERR
         DETACH -1             ..OK
         WRITE DS1,BUFFER2,ERROR=DISKERR
         DETACH -1             ..OK
         GOTO   DISK1
```

```
*                PRINT DISK ERROR MESSAGE
*
DISKERR  MOVE   ERROR,DISKTASK SAVE ERROR CODE
         ENQT                  GET CONTROL OF TERMINAL
         PRINTEXT TXT5
         PRINTNUM ERROR
         PRINTEXT  SKIP=1
         DEQT                  RELEASE TERMINAL
         ENDTASK 1              DETACH WITH CODE = 1
*
*
*                DATA AND CONSTANTS
*
TXT1     TEXT '@SAMPLE ANALOG DATA ACQUISITION PROGRAM@'
TXT2     TEXT '@ENTER RUN NUMBER '
TXT3     TEXT '@ENTER INTERVAL IN MS (10-10000) '
TXT4     TEXT '@ENTER NO. OF POINTS (10-10000) '
TXT5     TEXT '@DISK ERROR '
TXT6     TEXT '@RUN ENDED AFTER '
TXT7     TEXT ' POINTS@'
TXT9     TEXT '@READY FOR PI SIGNAL TO BEGIN TAKING DATA@'
TXT10    TEXT '@EXPERIMENT SWITCH = '
*
POINTCNT DATA  F'0'               NUMBER OF POINTS TAKEN
SWITCH   DATA  F'0'               SET TO '1' FOR 'ABORT'
RUNUM    DATA  F'0'               RUN IDENTIFIER
INTVL    DATA  F'0'               TIME INTERVAL
NPTS     DATA  F'0'               NUMBER OF POINTS TO TAKE
ERROR    DATA  F'0'
BUFFER   BUFFER 256,INDEX=BUFINDEX   DATA BUFFERS
BUFFER1  EQU    BUFFER         FIRST 128 WORDS
BUFFER2  EQU    BUFFER+256     SECOND 128 WORDS
*
         ENDPROG
         END
```

EXAMPLE 4: DIGITAL INPUT AND AVERAGING

This example illustrates the programming of a simple time averaging application. The program should read digital input group DI1 every time a process interrupt occurs on PI2. One complete scan is 128 data points. Each scan is to be added to a double-precision averaging buffer. The number of scans is read from the terminal as an initialization parameter. Also, the program asks whether to reset the averaging buffer before starting to scan. The maximum number of scans must be less than 1000.

```
START      GETVALUE  NSCAN,TXT1       GET NO. OF SCANS
           IF        (NSCAN,GE,1000),GOTO,ERROR
           RESET     PI2
           QUESTION  TXT2,NO=BEGIN    RESET AVERG. BUFFER?
           MOVE      ABUFR,0,256      YES - RESET IT
BEGIN      DO        NSCAN            SET UP FOR NSCANS
           DO        128              SET FOR 128 POINTS
           WAIT      PI2              WAIT FOR INTERRUPT
           RESET     PI2              RESET INTERRUPT
           SBIO      DI1,BUFR,INDEX     READ DI1(INDEXING)
           ENDDO
*
*         ONE SCAN COMPLETE - MOVE DATA TO AVERG BUFFER
*
           ADDV      ABUFR,BUFR,128,PREC=D
           MOVE      I,0             RESET BUFFER INDEX
           ENDDO
*
*          ALL SCANS COMPLETE
           PRINTEXT  TXT3
                .
                .
                .      THE REST OF THE PROGRAM

TXT1       TEXT      '@NUMBER OF SCANS - '
TXT2       TEXT      ' RESET AVERAGING BUFFER? '
TXT3       TEXT      ' ALL SCANS COMPLETE@'
NSCAN      DATA      F'0'
BUFR       BUFFER    128,INDEX=I
ABUFR      BUFFER    256
*
ERROR      PRINTEXT  TXT4       PRINT ERROR MESSAGE
           GOTO      START      RETURN FOR INPUT
TXT4       TEXT      ' TOO MANY SCANS - RE-ENTER@'
```

In this example, the number of scans to be done is read from the terminal and checked against 1000. If it is greater than or equal, an error message is printed and the program returns for a new input parameter. The operator is asked if the averaging buffer is to be reset. If yes, the MOVE instruction sets the averaging buffer (ABUFR) to 0. A loop is then initialized for the number of scans desired. A second loop is set up for a single scan of 128 points. The program waits for an interrupt on PI2 and, when it occurs, resets the interrupt for the next point, reads the digital input DI1 using automatic indexing into the buffer BUFR. When a scan is complete, the data is added to the ABUFR buffer. The buffer index, I, is reset to 0. When all scans are complete, a message is printed. The output from the program is illustrated in the following example.

```
NUMBER OF SCANS - 33
 RESET AVERAGING BUFFER? Y
 ALL SCANS COMPLETE
```

EXAMPLE 5: INDEX REGISTER USAGE

This example illustrates the use of the Event Driven Executive index registers. The two registers are symbolically referred to by #1 and #2. In this example, a vector BUFA, of length 1000, is to be compressed, removing all words equal to 0 and storing the compressed vector in the buffer 'BUFB'. When the buffer has been scanned, the length of the new buffer, BUFB, is to be printed on the terminal.

```
          MOVE     #1,0        SET REG 1 = 0
          MOVE     #2,0        SET REG 2 = 0
*
          DO       1000
          IF       ((BUFA,#1),EQ,0),GOTO,INCONE
          MOVE     (BUFB,#2),(BUFA,#1)
*
          ADD      #2,2        INCREMENT REG 2
INCONE    ADD      #1,2        INCREMENT REG 1
          ENDDO
*
          MOVE     NUMBERB,#2    SAVE BUFB LENGTH
          SHIFTR   NUMBER,1
          PRINTEXT MESSAGE
          PRINTNUM NUMBERB
          PRINTEXT SKIP=1
              .
              .
MESSAGE   TEXT     '@THE LENGTH OF BUFB = '
NUMBERB   DATA     F'0'
              .
              .
BUFA      BUFFER   1000
BUFB      BUFFER   1000
              .
              .
```

The example begins by initializing the two registers, #1 and #2. A DO loop is set up to scan the BUFA buffer of length 1000. If the value of BUFA is equal to 0, only the first register is incremented. Therefore, #1 is used to index through BUFA and #2 is used to index through the new buffer, BUFB. If the value of BUFA is non-zero, the data is moved to BUFB and both registers are incremented. When the scan is complete, the value of #2 is saved at the location NUMBERB and the message printed on the terminal. This program will display the following line on the terminal:

```
THE LENGTH OF BUFB =             2
```

EXAMPLE 6: USE OF MOVEA

This example shows the use of the MOVEA instruction in establishing addressability and indexability through a buffer. It is desired to average all values in the buffer and print the result.

```
          MOVEA     #1,BUFFER1           MOVE ADDRESS
          DO        256
          ADD       RESULT,(0,#1),PREC=D   SUM THE BUFFER
          ADD       #1,2
          ENDDO
          DIVIDE    RESULT,256,PREC=D
          PRINTEXT  '@AVERAGE VALUE OF ALL READINGS IS '
          PRINTNUM  RESULT,DWORD
          PRINTEXT  SKIP=1

*     . . .  CONTINUE PROGRAM

BUFFER1  BUFFER     256
RESULT   DATA       2F'0'       DOUBLE PRECISION

*     . . .  CONTINUE PROGRAM
```

In this example the address of the buffer, BUFFER1 is moved into register #1. The DO loop is entered, and for each pass through the loop, register #1 is incremented to the next word. RESULT is declared as 2 words, the ADD has a PREC=D parameter in order to hold the sum. After the division, RESULT is printed. The output from this program is illustrated in the following example.

```
AVERAGE VALUE OF ALL READINGS IS          1
```

EXAMPLE 7: A TWO TASK PROGRAM WITH ATTNLIST

The preceding examples illustrate the use of many of the Event Driven Executive instructions. This example is given to illustrate a program containing two simultaneously executing tasks and the ATTNLIST statement being used for operator control.

The problem is to count the number of process interrupts occurring on process interrupt PI1 for an extended period, printing the total number recorded on the terminal every minute. In addition, the program must be started, stopped, and restarted from the terminal. The complete program follows:

```
*    COUNT PROCESS INTERRUPTS
*
TASK1     PROGRAM   TABL1,100
          ATTNLIST (RUN,START,STOP,STOPIT)  INPUT CODES
*         ATTENTION ROUTINES
START     MOVE      PICOUNT,0       SET PI COUNTER=0
          MOVE      MINCOUNT,0      SET MINUTES=0
          POST      RUNECB          START RUN
          ENDATTN                   RETURN TO SUPERVISOR
*
STOPIT    MOVE      SWITCH,1        SET STOP SWITCH
          ENDATTN
*         DATA AND TEXT MESSAGES
PICOUNT   DATA      F'0'            PI COUNTER
MINCOUNT  DATA      F'0'            MINUTES
COUNTS    DATA      F'0'            SAVE WORD
SWITCH    DATA      F'0'            STOP SWITCH
RUNECB    ECB
TXT1      TEXT      ' NUMBER OF INTERRUPTS AFTER '
TXT2      TEXT      ' MINUTES - '
*
*    TASK 1 -  PRINT NUMBER OF INTERRUPTS
*              EVERY MINUTE ON TERMINAL
TABL1     ATTACH    TASK2           START TASK 2
          RESET     RUNECB          RESET THE RUN EVENT
AWAIT     WAIT      RUNECB          WAIT FOR START CODE
          STIMER    60000           SET TIMER FOR 1 MIN
          MOVE      COUNTS,PICOUNT  SAVE PICOUNT
          ENQT      $SYSPRTR
```

```
         PRINTEXT  TXT1            PRINT PI COUNTS
         PRINTNUM  MINCOUNT        AND MINUTES
         PRINTEXT  TXT2
         PRINTNUM  COUNTS
         PRINTEXT  SKIP=1
         DEQT
         WAIT      TIMER           WAIT FOR TIMER
         ADD       MINCOUNT,1      INCREMENT MINUTES
         IF        (SWITCH,EQ,0),GOTO,AWAIT
         ENQT      $SYSPRTR
         PRINTEXT  '@ COUNT PROCESS INTERRUPTS ENDING@'
         DEQT
         PROGSTOP
*    TASK 2 -  WAIT FOR A PROCESS INTERRUPT AND
*              INCREMENT THE COUNTER
*
TASK2    TASK      TABL2,10
*
TABL2    STIMER    15000,WAIT    SET TIMER FOR 15 SECONDS
*                                AND WAIT FOR INTERRUPT
         ADD       PICOUNT,1     INCREMENT COUNTER AND
         GOTO      TABL2         RETURN TO WAIT
         ENDTASK
         ENDPROG                 END OF PROGRAM
         END
```

PROGRAM names the primary task, TASK1, gives the label of the first instruction, TABL1, and defines the priority of the primary task as 100. Keying RUN or STOP causes the user program to be entered at START or STOPIT, respectively. and executed under the ATTNLIST task. START resets the interrupt and minute counters and releases TASK1 by posting the event RUNECB.

TASK1 is started automatically by the system. It starts TASK2 via the ATTACH instruction. TASK2 starts at the instruction with label TABL2 and has a priority of 10. The event RUNECB is reset and the program issues a WAIT for the event. TASK1 is now suspended until RUN is keyed. When the event is posted, the program sets a timer for 60000 milliseconds (1 minute). The number of interrupts is saved in COUNTS. The terminal is enqueued, the message printed, the terminal dequeued, the minute counter is incremented, and the program waits for the next interval. If, during the time period, STOP was keyed, the program will print a termination message and terminate. TASK2 sets a timer interrupt for 4 seconds and waits on the interrupt, increments the counter, and returns to wait for the next interrupt. This will continue indefinitely.

This example illustrates the use of parallel running tasks and the possibilities for operator control and interaction.

EXAMPLE 8: PROGRAM LOADING FUNCTIONS

The following program illustrates the process of one program loading another with the LOAD instruction. The program TEST1 prints two opening messages separated by 2 blank lines, loads the program TEST2, tests for a successful LOAD and then WAITs for the loaded program to end. This illustrates how programs can be synchronized.

```
TEST1     PROGRAM START1
*
START1     PRINTEXT   '@TEST PROGRAM LOADING@@'
AGAIN      PRINTEXT   '@HELLO - TEST1 WILL LOAD TEST2@'
*
*
           LOAD       TEST2,LOGMSG=NO,EVENT=EV1,ERROR=STOP1
*
           WAIT       EV1      WAIT FOR TEST2 TO END
*
           QUESTION  '@END THE PROGRAM (Y OR N) ?',NO=AGAIN
STOP1      PROGSTOP
EV1        ECB
           ENDPROG
           END
```

This is the program to be loaded, TEST2. It can also be loaded independently from a terminal. A message is printed, the program waits 5 seconds, prints again, and ends; TEST1 is notified by the supervisor that TEST2 has ended.

```
TEST2      PROGRAM    START2
START2     PRINTEXT   '@TEST2 HERE, I WILL DELAY 5 SECONDS@'
           STIMER     5000,WAIT
           PRINTEXT   '@TIME IS UP, RETURNING TO TEST1@@'
           PROGSTOP   LOGMSG=NO
           ENDPROG
           END
```

EXAMPLE 9: FLOATING POINT, WAIT/POST, GETEDIT/PUTEDIT

The program prompts the user for two numbers, each can be up to 20 digits, with or without decimal points. It then performs floating-point addition, subtraction, multiplication, and division, and prints the results in floating-point notation with up to 14 digits after the decimal point.

The use of the GETEDIT and PUTEDIT instructions using formatting are illustrated, as well as WAIT and POST and floating point arithmetic.

```
FPDEMO PROGRAM   START,FLOAT=YES
*
ATTNLIST ATTNLIST  (STOP,POST1,CALC,POST2)
*
POST1    POST      KBEVENT,1
         ENDATTN
*
POST2    POST      KBEVENT
         ENDATTN
START    EQU       *
LOOP     EQU       *
         PRINTEXT 'PRESS "ATTN" ENTER "CALC" OR "STOP" @'
         WAIT      KBEVENT,RESET  WAIT TILL CALC ENTERED
         IF        KBEVENT,NE,-1,STOP  GO TO STOP IF
*                                      STOP ENTERED
*
READA    EQU       *
         PRINTEXT  'A = ',SKIP=2
         GETEDIT   FMT1,WORK,((A,1,L)),SCAN=FREE  GET A
*
READB    EQU       *
         PRINTEXT  'B = ',SKIP=2
         GETEDIT   FMT1,WORK,((B,1,L)),SCAN=FREE  GET B
```

```
LIST     EQU      *
         PRINTEXT '@A + B = '
         FADD     A,B,RESULT=C,PREC=LLL
         PUTEDIT  FMT2,WORK,((C,1,L))        PRINT A+B
*
         PRINTEXT '@A - B = '
         FSUB     A,B,RESULT=C,PREC=LLL
         PUTEDIT  FMT2,WORK,((C,1,L))        PRINT A-B
*
         PRINTEXT '@A * B = '
         FMULT    A,B,RESULT=C,PREC=LLL
         PUTEDIT  FMT2,WORK,((C,1,L))        PRINT A*B
*
         PRINTEXT '@A / B = '
         FDIVD    A,B,RESULT=C,PREC=LLL
         PUTEDIT  FMT2,WORK,((C,1,L))        PRINT A/B
*
         PRINTEXT SKIP=1
         GOTO     LOOP
*
STOP     EQU      *
         PROGSTOP
*
A        DC       2D'0'
B        DC       2D'0'
C        DC       2D'0'
D        DC       2D'0'
WORK     TEXT     LENGTH=20
FMT1     FORMAT   (F20.0)
FMT2     FORMAT   (E20.14)
KBEVENT  ECB
         ENDPROG
         END
```

EXAMPLE 10: USER EXIT ROUTINE

These examples (actual code from the Event Driven Executive) illustrate:

1. How an instruction can be added to the Event Driven Executive Macro Libraries by the user, using the USER instruction.

2. How a user exit routine is structured.

The following macro definition illustrates how the user who understands assembler coding can create his own Event Driven Executive instructions using macros and the Event Driven Executive USER instruction.

The SQRT macro call in the programming example is described under "SQRT" on page 277.

```
LABEL    SQRT    rsq, root, rem
```

is converted by the following macro definition (in MACLIB) and the Series/1 Macro or Host Assembler.

```
                MACRO
 &LABEL         SQRT     &RSQ,&ROOT,&REM,&P1=,&P2=,&P3=
                GBLB     &SQRT
                AIF      (N'&SYSLIST NE 3).ER1
 &LABEL         USER     SQRT,PARM=(&RSQ,&ROOT,&REM),        C
                         P=(&P1,&P2,&P3)
                AIF      (&SQRT).DONE
                GOTO     $SQ&SYNDX
                $SQRT
 $SQ&SYNDX      EQU      *
 .DONE          ANOP
 &SQRT          SETB     1
                MEXIT
 .ER1           ANOP
                MNOTE    8,'** NUMBER OF OPERANDS NOT 3 **'
                MEND
```

to:

```
 LABEL          USER     SQRT,PARM=(RSQ,ROOT,REM)
                GOTO     $SQnn    -  On 1st occurrence
                $SQRT             -  of SQRT instruction
 $SQnn          EQU  *            -  only
```

where $SQRT is used to include the actual user exit routine (SQRT) which calculates the square root. This routine could have been explicitly stated in the macro definition where $SQRT is coded, or, as in this case, brought in from the macro library where it was stored as the macro definition $SQRT. This technique for including the user exit routine relieves the end user of the need to know whether the routine has or has not been included in his program.

The user exit routine SQRT which is brought into the user program when $SQRT is encountered illustrates the considerations which are noted under USER instruction description.

```
SQRT    EQU    *              SQUARE ROOT ROUTINE
*
        MVD    (R1)*,R3       LOAD VALUE
SQ00    MVW    R3,R6          SAVE HIGH ORDER
        MVWI   X'8000',R5     PUT CONSTANT IN R5
        SRL    14,R3          CHECK INPUT FOR TOO LARGE
        JZ     SQ01           IF ZERO ITS IN RANGE
        J      SQ07           IF NOT, BACK TO CALLER
SQ01    AW     R5,R3          ASSUME NEXT BIT IS A 1
        IR     R6,R3          SWAP ROOT AND REMAINDER
        SLL    1,R3           MPY REM BY 2
        SLC    1,R4           MPY REM LOW ORDER BY 2
        JCY    SQ06           NEXT ROOT BIT IS A 0
        JEV    SQ01A          SKIP UNLESS LOW ORDER OF LW
        ABI    1,R3           ADD CARRY TO HI ORDER OF LW
SQ01A   SW     R6,R3          SUB TRIAL ROOT FROM REM
        JCY    SQ05           GO FIX REM
SQ02    IR     R6,R3          SWAP REM AND TRIAL ROOT
        AW     R5,R3          DOUBLE DIGIT FOR NEXT PASS
SQ03    SRL    1,R5           HALF ADJUST FACTOR
        JNZ    SQ01           NOT DONE, GO AGAIN
        SRL    1,R3           CORRECT ROOT
SQ04    ABI    2,R1           POINT TO ROOT SAVE ADDR
        MVW    R3,(R1)*       SAVE ROOT
        ABI    2,R1           POINT TO REM SAVE ADDR
        MVW    R6,(R1)*       SAVE REM
        ABI    2,R1           POINT TO NXT INSTR
        BX     RETURN         SWITCH BACK TO EDL
        AW     R6,R3          CORRECT REM
        IR     R6,R3          SWAP REM AND ROOT
        SW     R5,R3          SET THIS DIGIT TO ZERO
        J      SQ03           GO SET UP FOR NEXT PASS
SQ06    JEV    SQ06A          SKIP UNLESS LOW WORD
        ABI    1,R3           ADD CARRY TO HI ORDR WD
SQ06A   SW     R6,R3          SUB ROOT FROM REM
        J      SQ02           GO SET UP FOR NXT PASS
SQ07    MVBI   0,R3           ZERO ROOT
        MVBI   0,R6           ZERO REM
        J      SQ04           GO SET UP FOR EXIT
```

1. The SQRT EQU * statement defines the entry point for the USER instruction generated above.

2. On entry, R1 points to the location where the address of the first parameter is stored. The first instruction moves the double word (VALUE) to register 3 and 4.

3. At location SQ04, R1 is incremented by 2 to point to the location where the address of the second parameter (ROOT) is stored. Two lines lower, at the ABI instruction, R1 is again incremented to point to the location where the address of the third parameter (REM) is stored.

4. Two lines lower, R1 is again incremented by 2 to point to the return address - the Event Driven instruction following the USER instruction.

5. At the line prior to SQ05, the routine branches back to the user.

6. As required, R2 has not been changed by the routine.

EXAMPLE 11: I/O LEVEL CONTROL PROGRAM

This program illustrates the use of EXIO control functions to provide your own support for an I/O device. Its use would require definition of the EXIO devices by including statements similar to the following in the 'System Configuration' statements:

```
EXIODEV    E0,MAXDCB=1
EXIODEV    E4,MAXDCB=3,RSB=6,END=YES
```

The devices to be controlled are the controller and one line of PCS (IBM 4987, Programmable Communication Subsystem). The program prepares both devices to interrupt and loads controller storage.

```
LDPCS      PROGRAM    PSTART
*
*  Attach Interrupt Handler Tasks
*
These tasks will wait until the EXIO interrupt handler
posts an appropriate ECB.  They will then service that
particular interrupt.
*
PSTART     ATTACH     DEINT        HANDLES DEVICE END
           ATTACH     EXCINT     HANDLES OTHER INTERRUPTS
*
*
*  Place a User List Address in the Device Descriptor Block
*
*  PCSLIST points to a list of 3 addresses used by the EXIO
*  interrupt handler:
*
PCSID      STORES 3 WORDS DESCRIBING THE INTERRUPT
PCSECB     A LIST OF ECBS
PCSSDCB    A DCB USED TO START CYCLE STEAL STATUS
*
*
           EXOPEN     E0,PCSLIST,ERROR=OPNERR
           EXOPEN     E4,PCSLIST,ERROR=OPNERR
```

```
*   Prepare the Controller to Interrupt
*
*   The instruction points to the IDCB 'PRPIDCB0' which
*   describes an IO operation which will prepare the device
*   at address E0 to interrupt on hardware level 1.  If the
*   IO instruction is not accepted, execution will resume at
*   'PREPERR'.
*
*
           EXIO       PRPIDCB0,ERROR=PREPERR
*
*   Prepare Line 4 to Interrupt
*
           EXIO       PRPIDCB4,ERROR=PREPERR
*
*
*   Load PCS Controller Storage
*
*   The IDCB points to a DCB, 'LDDCB', which describes an IO
*   operation which will load the controller with the data at
*   'PSCORE'.
*
*
           EXIO       LDIDCB,ERROR=LDERR
*
*
*   Wait for the Load to Complete
*
*   This will be indicated by the posting of the ECB
*   'DONECB'.  'DONECB' will be posted by the interrupt
*   handler task 'DEINT'.  The task 'DEINT' will execute when
*   the ECB 'PDEECB' is posted.  'PDEECB' will be posted by
*   the EXIO interrupt handler when an interrupt with a ccode
*   3 (device end) is received.
*
*
           WAIT       DONECB
PREND      PROGSTOP
```

```
*   Enter here if EXOPEN instruction executes with error
*
*
OPNERR     MOVE      CC,LDPCS
           ENQT
           PRINTEXT  '@EXOPEN REJECTED, CC = '
           PRINTNUM  CC,MODE=HEX
           DEQT
           GOTO      PREND
*
*
*   Enter here if Prepare Command is not accepted
*
*
PREPERR    MOVE      CC,LDPCS
           ENQT
           PRINTEXT  '@PCS PREPARE EXIO REJECTED, CC = '
           PRINTNUM  CC,MODE=HEX
           DEQT
           GOTO      PREND
*
*
*   Enter here if LOAD command is not accepted
*
*
LDERR      EQU       *
           MOVE      CC,LDPCS
           ENQT
           PRINTEXT  '@LOAD EXIO REJECTED, CC = '
           PRINTNUM  CC,MODE=HEX
           DEQT
           GOTO      PREND
*
*
*   Execute if interrupt other than 'Device End' is received
*
*
EXCINT     TASK      EXCSTART
EXCSTART   WAIT      PEXCECB,RESET
           AND       PEXCECB,X'7FFF'
           ENQT
           PRINTEXT  '@ INTERRUPT, CCODE & DEV ADR = '
           PRINTNUM  PEXCECB,MODE=HEX
           DEQT
           POST      DONECB,2
           ENDTASK
```

```
*  Execute if 'Device End' interrupt is received
*
*
DEINT        TASK        DESTART
DESTART      WAIT        PDEECB,RESET
             ENQT
             PRINTEXT    '@PCS CONTROL STORAGE LOADED@'
             DEQT
             POST        DONECB,-1
             ENDTASK
*
*
*  Define where information is to be stored after EXIO
*  device interrupt
*
*
PCSLIST      DATA        A(PCSID)
             DATA        A(PCSECB)
             DATA        A(PCSSDCB)
*
*
*  Will Receive:  Interrupt ID Word, LSR, ADDR of ECB Posted
*
*
PCSID        DATA        3F'0'
*
*
*  Addresses of ECB's to be posted
*
*
PCSECB       DATA        A(PEXCECB)     CONTROLLER END
             DATA        A(PEXCECB)     PCI
             DATA        A(PEXCECB)     EXCEPTION
             DATA        A(PDEECB)      DEVICE END
             DATA        A(PEXCECB)     ATTENTION
             DATA        A(PEXCECB)     ATTN + PCI
             DATA        A(PEXCECB)     ATTN + EXC
             DATA        A(PEXCECB)     ATTN + DE
*
PEXCECB      ECB         0
PDEECB       ECB         0
DONECB       ECB         0
```

```
*   This DCB will be used to start Cycle Steal Status if an
*   interrupt is received.
*
*
PCSSDCB     DCB       IOTYPE=INPUT,COUNT=10,DATADDR=CSSDATA
CSSDATA     DATA      5F'0'       CYCLE STEAL STATUS DATA
CC          DATA      F'0'
*
PRPIDCB0    IDCB      COMMAND=PREPARE,ADDRESS=E0
PRPIDCB4    IDCB      COMMAND=PREPARE,ADDRESS=E4
LDIDCB      IDCB      COMMAND=START,ADDRESS=E0,
                                    MOD4=6,DCB=LDDCB
LDDCB       DCB       DVPARM1=0200,COUNT=PCSLCNT,
                                    DATADDR=PCSCORE
*
*   PCS Controller Storage Load
*
PCSCORE     EQU       *
            DC        64F'0'
            DC        X'9284',X'928A',X'0001',X'9352',X'0000'
            DC        X'0001',X'928F',X'0001'
            DC        97F'0'
            DC        X'0005',X'935E',X'0000',X'0000',X'0000'
            DC        X'936A',X'0800',X'0400',X'5B00',X'9368'
            DC        X'4800',X'A220',X'0700',X'0501',X'3D04'
            DC        X'937A',X'A204',X'0002',X'0101',X'0000'
            DC        7F'0'
            DC        X'0003',X'0D07',X'0A00'
PCSEND      EQU       *
PCSLCNT     EQU       PCSEND-PCSSCORE
            ENDPROG
            END
```

EXAMPLE 12: GRAPHICS INSTRUCTIONS PROGRAMMING EXAMPLE

In the following example the graphic control characters (GS, US, ESC, etc.) are assumed to have certain meanings for the terminal. A different terminal may require the use of different control characters to perform a similar functions.

The example illustrates the use of the graphics instructions described on the preceding pages. This program will print a message, plot a curve with axes, put the cross-hair on the screen, wait for the user to position the cross-hair and depress a key and carriage return, and then display the character entered and x,y coordinates of the cross-hair position. The user may then end the program or start it again.

The program starts at the label START where a short message is printed. The text string character count is reset, and the ESC code is put into TEXT1, followed by the FF character. The sequence ESC FF will erase the screen and send the alpha cursor to the home position (upper left corner). The PRINTEXT instruction will cause this to occur. Now, depending on the type of terminal and the line speed, it may be necessary to delay for a second to allow the erase sequence to complete. This is accomplished by the STIMER instruction. The text string is reset again and the graph mode character, GS, is added to the text string. The SCREEN instruction is used to form the 4 characters required to draw a dark vector to the screen address (520,300). The 4 characters represent the Hi Y, Lo Y, Hi X, and Lo X values. To write an axis label at this position, it is necessary to return to alpha mode. This requires the US character. The two PRINTEXT instructions are executed to perform the full operation. Note XLATE=NO on PRINTEXT prevents conversion of data as it is already in ASCII.

Now the data, YDATA (8 points), is plotted using the YTPLOT instruction. The plot area and coordinates are given by the 8 words at the label PCB. The plot area in screen addresses is 500 to 1000 in the x-direction (horizontal) and 100 to 600 in the y-direction (vertical). The corresponding plot area in the user's coordinates is 0 to 10 in the x-direction and -5 to 5 in the y-direction. The X and Y axes are drawn by the next two XYPLOT instructions. Each of these is simply a 2-point plot, from the origin to the end point. The cross-hair cursor is now put on the screen by the PLOTGIN instruction. The user should position the cursor and enter a character. When the character is received, the cursor position is converted to the plot coordinates as specified at PCB, and the results are stored at X and Y. The next few instructions print out the results of this action and ask if the user wishes to end the program.

```
         PRINT       NOGEN
GTEST    PROGRAM     START
START    EQU         *
         PRINTEXT    'GRAPHICS TEST PROGRAM PRESS ENTER @'
         READTEXT    TEXT1
         CONCAT      TEXT1,ESC,RESET
         CONCAT      TEXT1,FF
         PRINTEXT    TEXT1,XLATE=NO
         STIMER      1000,WAIT
         CONCAT      TEXT1,GS,RESET
         SCREEN      TEXT1,520,300,CONCAT=YES
         CONCAT      TEXT1,US
         PRINTEXT    TEXT1,XLATE=NO
         PRINTEXT    TEXT3
         YTPLOT      YDATA,X1,PCB,NPTS,1
         XYPLOT      YAXISX,YAXISY,PCB,TWO
         XYPLOT      XAXISX,XAXISY,PCB,TWO
         PLOTGIN     X,Y,CHAR,PCB
         PRINTEXT    TEXT4
         PRINTEXT    CHAR,XLATE=NO
         PRINTEXT    TEXT5
         PRINTNUM    X,2
         QUESTION    TEXT6,NO=START
         PROGSTOP
TEXT1    TEXT        LENGTH=30
TEXT3    TEXT        'X-AXIS LABEL'
TEXT4    TEXT        '@CHARACTER STRUCK WAS '
TEXT5    TEXT        '@X,Y COORDINATES ='
TEXT6    TEXT        '@END PROG (Y/N)? '
         DATA        X'0201'
CHAR     DATA        F'0'
YDATA    DATA        F'0'
         DATA        F'1'
         DATA        F'0'
         DATA        F'2'
         DATA        F'0'
         DATA        F'1'
         DATA        F'-2'
         DATA        F'-1'
X1       DATA        F'0'
NPTS     DATA        F'8'
YAXISX   DATA        2F'0'
YAXISY   DATA        F'-5'
```

```
         DATA     F'5'
XAXISX   DATA     F'0'
         DATA     F'10'
XAXISY   DATA     2F'0'
TWO      DATA     F'2'
PCB      DATA     F'500'
         DATA     F'1000'
         DATA     F'0'
         DATA     F'10'
         DATA     F'100'
         DATA     F'600'
         DATA     F'-5'
         DATA     F'5'
X        DATA     F'0'
Y        DATA     F'0'
         ENDPROG
         END
```

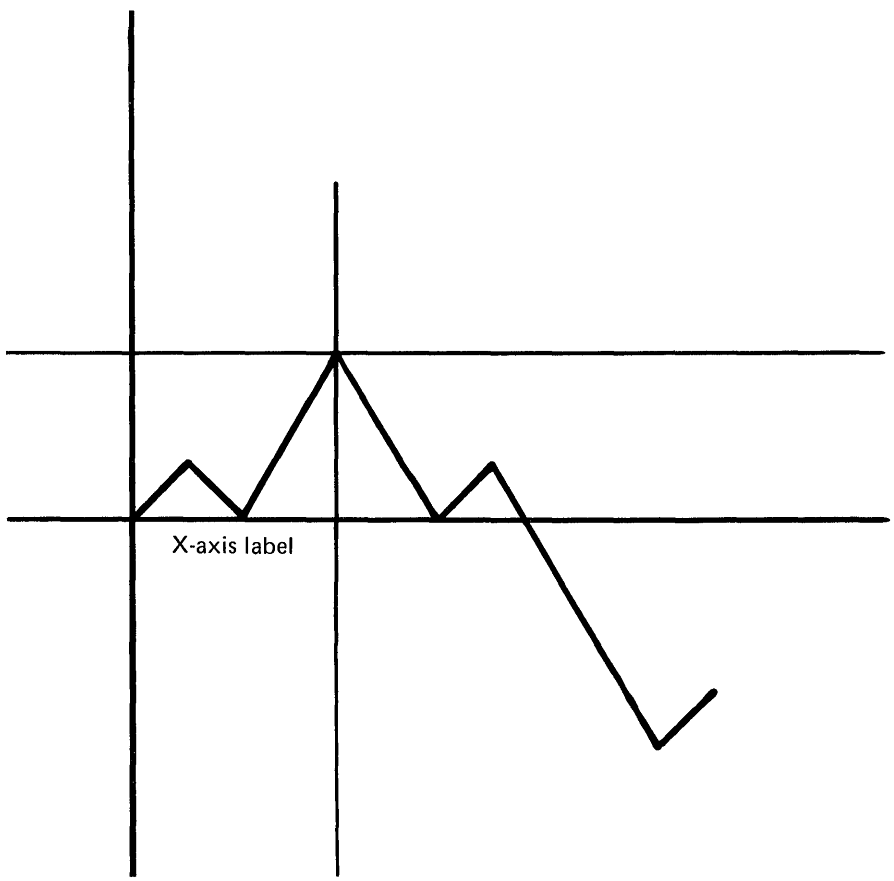


Figure 19. Graphic Program Output: This figure shows the result of the preceding program.

EXAMPLE 13: FORMAT AND DISPLAY TRACE DATA

This program formats and displays the contents of the software trace table. The first entry displayed is the one that was most recently entered. The user is requested to enter the hexadecimal address of the trace table. Sample output is shown following the source code.

```
$FORMAT PROGRAM  START
START   EQU      *
        PRINTEXT 'ENTER CIRCBUFF ENTRY POINT: ',LINE=0
        GETVALUE CIRENTRY,MODE=HEX
        MOVE     #1,CIRENTRY                #1 = A(TRACE TBL)
        PRINTEXT 'MACHINE/PROGRAM CHECK STATUS REPORT',LINE=0
        PRINTEXT SKIP=3
        PRINTEXT 'SINCE IPL '
        MOVE     CIRCNT,(+$CIRCNT,#1)
        PRINTNUM CIRCNT,TYPE=S,FORMAT=(5,0,I)
        PRINTEXT ' STATUS ENTRIES HAVE BEEN RECORDED'
        PRINTEXT SKIP=2
        MOVE     #2,(+$CIRSTR,#1)           #2 = A(FIRST ENTRY)
        SUB      (+$CIREND,#1),#2,RESULT=BYTESIZE
        DIVIDE   BYTESIZE,(+$CIRESIZ,#1),RESULT=ENTRYCNT
        IF       (CIRCNT,NE,0)       IF THERE WERE ENTRIES
                 PRINTEXT HEADING
                 PRINTEXT SKIP=2
                 MOVE #2,(+$CIRIN,#1)       #2 = A(NEXT ENTRY)
                 DO   ENTRYCNT,TIMES
                      SUB #2,(+$CIRESIZ,#1) #2 = A(PREV ENTRY)
                      IF  #2,LT,(+$CIRSTR,#1)
                          MULT (+$CIRESIZ,#1),ENTRYCNT,
                               RESULT=NUMBER
                          SUB  NUMBER,(+$CIRESIZ,#1)
                          ADD  (+$CIRSTR,#1),NUMBER,RESULT=#2
                      ENDIF
                      IF  (+$CIRPSW,#2),EQ,0
                          IF ((+$CIRLSB,#2),EQ,0),GOTO,FINISH
                      ENDIF
                      MOVE     NUMBER,(+$CIRSTAT,#2)
                      PRINTNUM NUMBER,MODE=HEX    ST VAR/EAK
                      PRINTNUM (+$CIRTCBA,#2),MODE=HEX   A(TCB)
                      PRINTNUM (+$CIRPSW,#2),MODE=HEX    PSW
                      PRINTNUM (+$CIRSAR,#2),MODE=HEX      SAR
                      PRINTNUM (+$CIRLSB,#2),11,MODE=HEX   LSB
                      PRINTEXT SKIP=1
                 ENDDO
                 GOTO FINISH
        ENDIF
        PRINTEXT 'NO ENTRIES TO DUMP'
FINISH  EQU      *
        PROGSTOP
```

```
BYTESIZE DATA     F'0'          SIZE OF TRACE TABLE ENTRY SPACE
ENTRYCNT DATA     F'0'          # OF ENTRIES IN TABLE FOR DISPLAY
LOCATION DATA     F'0'          LOCATION POINTER
NUMBER   DATA     F'0'          NUMERIC WORK WORD
CIRENTRY DATA     F'0'          TRACE TABLE ENTRY POINT
CIRCNT   DATA     F'0'          # OF ENTRIES IN BUFFER
HEADING  TEXT     'S/EAK TCBA  PSW  SAR  IAR  AKR  LSR  0  1  2  X
                3  4  5  6  7'
$CIRSTR  EQU      0
$CIRIN   EQU      $CIRSTR+2
$CIREND  EQU      $CIRIN+2
$CIRCNT  EQU      $CIREND+2
$CIRESIZ EQU      $CIRCNT+2
$CIRESTR EQU      $CIRESIZ+2
$CIRSTAT EQU      0
$CIREAK  EQU      $CIRSTAT+1
$CIRTCBA EQU      $CIRSTAT+2
$CIRPSW  EQU      $CIRTCBA+2
$CIRSAR  EQU      $CIRPSW+2
$CIRLSB  EQU      $CIRSAR+2
         ENDPROG
         END
```

```
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

```
SAMPLE   PROGRAM START,DS=??,ERRXIT=TEECB
START    EQU   *
*
*
         ENQT
         PRINTEXT LOGON,LINE=0  PRINT LOGON MESSAGE
         DEQT
*
*  OPEN the Indexed Access Method data set for loading
*
        CALL  IAM,(LOAD),IACB,(DS1),(OPENTAB),(SHARE)
*
*  LOAD the Indexed Access Method data set
*
        MOVEA POINTER,RECORD1          POINTER <== A(RECORD1)
        DO    RECNUM,TIMES
              CALL  IAM,(PUT),IACB,(*),P4=POINTER
              ADD POINTER,80           POINT TO NEXT RECORD
        ENDDO
*    GET OUT OF LOAD MODE
        CALL  IAM,(DISCONN),IACB
        EJECT
*
*  OPEN the indexed file for processing
*
        CALL  IAM,(PROCESS),IACB,(DS1),(OPENTAB),(SHARE)
*
*
* Perform a direct retrieval of the first record whose key is
* greater than 'JONES PW'.  The key field will be modified to
* reflect the key of the record retrieved.
*
        CALL  IAM,(GET),IACB,(BUFF),(KEY3),(GT)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
*
*  Perform a sequential retrieval of the first two records
*  whose keys are greater than or equal to 'JONES PW'
*
        CALL  IAM,(GETSEQ),IACB,(BUFF),(KEY1),(GE)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
        CALL  IAM,(GETSEQ),IACB,(BUFF)
        MOVE  RTCODE,SAMPLE
        IF    (SAMPLE,NE,-1),GOTO,IAMERR
        CALL  IAM,(ENDSEQ),IACB,(BUFF)    END SEQUENTIAL MODE
```

```
*
*   Update the record whose key is 'JONES PW' by a
*   direct update
*
         CALL   IAM,(GET),IACB,(BUFF),(KEY1),(UPEQ)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
*
*   Make the desired modifications to the record now in BUFFER
*
*        MOVE   BUFF+30,0
         CALL   IAM,(PUTUP),IACB,(BUFF)
*
*   Delete the record whose key is 'JONES PW' by a
*   sequential update
*
         CALL   IAM,(GETSEQ),IACB,(BUFF),(KEY1),(UPEQ)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
         CALL   IAM,(PUTDE),IACB,(BUFF)
         CALL   IAM,(ENDSEQ),IACB            END SEQUENTIAL MODE
*
*   Insert a new record with a key of 'MATHIS GR'
*
         CALL   IAM,(PUT),IACB,(NEWREC)
*
*   Delete the record whose key is 'LANG LK'
*
         CALL   IAM,(DELETE),IACB,(KEY2)
         MOVE   RTCODE,SAMPLE
         IF     (SAMPLE,NE,-1),GOTO,IAMERR
         EJECT
*
*   Extract the file control block into the extract buffer
*
         CALL   IAM,(EXTRACT),IACB,(EXTBUF),(FCBSIZE),128
         MOVEA  #1,EXTBUF        #1  <--  A(EXTRACT BUFFER)
         MOVE   FLAGBYTE,(0,#1),BYTE   OBTAIN FCB FLAG BYTE
         SPACE 5
*
*   Write verification complete message to the operator
*
         ENQT
                PRINTEXT SKIP=1
                PRINTEXT VERIF,SPACES=0
         DEQT
         GOTO   FINISH       JUMP AROUND ERROR ROUTINES
SYSERR   EQU    *            GETS CONTROL ON SYSIPGM CHECK
```

```
* When a task error exit is specified in an Indexed
* Access Method program, you can release all active
* record and block level locks as well as disconnect
* the file itself issuing the 'DISCONN' call for each
* file that is open.
*
        GOTO  FINISH
        EJECT
IAMERR  EQU   *                    GETS CONTROL UPON INDEXED
*                                  METHOD ERRORS
        MOVE  RTCODE,SAMPLE
        ENQT
              PRINTEXT SKIP=2
              PRINTEXT RTCODMSG
              PRINTNUM RTCODE,TYPE=S,FORMAT=(3,0,I)
              PRINTEXT SKIP=1
              PRINTEXT ERRMSG,SPACES=0
        DEQT
FINISH  EQU   *
        CALL  IAM,(DISCONN),IACB
        PROGSTOP
        EJECT
*
*  Data definition and storage areas
*
RECNUM   DATA   F'10'          NUMBER OF RECORDS TO LOAD
RTCODE   DATA   F'0'      INDEXED ACCESS METHOD RETURN CODE
OPENTAB  DATA   F'0'           SYSTEM RETURN CODE ADDRESS
         DATA   A(IAMERR)      ERROR EXIT ROUTINE ADDRESS
         DATA   F'0'           END OF DATA ROUTINE ADDRESS
RECORD1  DATA   CL80'BAKER RG'
RECORD2  DATA   CL80'DAVIS EN'
RECORD3  DATA   CL80'HARRIS SL'
RECORD4  DATA   CL80'JONES PW'
RECORD5  DATA   CL80'JONES TR'
RECORD6  DATA   CL80'LANG LK'
RECORD7  DATA   CL80'PORTER JS'
RECORD8  DATA   CL80'SMITH AR'
RECORD9  DATA   CL80'SMITH GA'
RECORD10 DATA   CL80'THOMAS SN'
FLAGBYTE DATA   H'0'                      FCB FLAG BYTE
         DATA   H'0'
```

```
NEWREC   DATA   CL80'MATHIS GR'
BUFF     DATA   CL80' '
         DATA   X'1C'                  TOTAL LENGTH OF KEY
         DATA   X'00'                  USE ALL OF THE KEY
KEY1     DATA   CL28'JONES PW'
         DATA   X'1C'
         DATA   X'00'
KEY2     DATA   CL28'LANG LK'
         DATA   X'1C'
         DATA   X'00'
KEY3     DATA   CL28'JONES PW'
IACB     DATA   F'0'               ADDR OF IACB PUT HERE
EXTBUF   DATA   64F'0'             FCB PUT HERE BY EXTRACT
LOGON    TEXT   'INSTALLATION VERFICATION PROGRAM ACTIVE'
VERIF    TEXT   'VERIFICATION COMPLETE'
ERRMSG   TEXT   'VERIFICATION INCOMPLETE DUE TO BAD RETURN CODES'
RTCODMSG TEXT   'INDEXED ACCESS METHOD RETURN CODE: '
         EJECT
*
*  The following storage is used by task error exit handling
*
TEECB    EQU   *           TASK ERROR EXIT CONTROL BLOCK
         DATA  F'2'        # OF DATA WORDS THAT FOLLOW
         DATA  A(SYSERR)   ADDRESS OF EXIT ROUTINE
         DATA  A(HSA)      ADDRESS OF HARDWARE STATUS AREA
*  Hardware status area.  This storage will be filled in by
*  hardware upon system or program check interrupt
HSA      EQU   *
         DATA  F'0'          PROCESSOR STATUS WORD
HSALSB   EQU   *             LEVEL STATUS BLOCK:

         DATA  F'0'             ADDRESS KEY REGISTER
         DATA  F'0'             INSTRUCTION ADDR REGISTER
         DATA  F'0'             LEVEL STATUS REGISTER
         DATA  8F'0'            GENERAL REGISTERS 0-7
         COPY  IAMEQU
         COPY  FCBEQU
         ENDPROG
         END
```

EXAMPLE 15: WRITE DATA TO TAPE DATA SET

This example generates a 300 byte record using a DATA statement. The record consists of the word TEST, repeated 75 times. The record is then written to a tape data set that is named by you when prompted by the PROGRAM statement. Any tape related error condition will print a return code (RC) using the PRINTEXT statement at location ERR. If no errors occur, after 300 records have been written the tape data set will be closed, the tape will be rewound and the tape drive will be placed in an off-line status by the CONTROL statement at location ENDIT.

```
TEST    PROGRAM    START,DS=(??)
*
*
START   EQU   *
        PRINTEXT   '@BEGIN TEST PROGRAM@'
*
*
        DO         300,TIMES
        WRITE      DS1,BUFF,1,300,ERROR=ERR,WAIT=YES
        ENDDO
*
*
ENDIT   EQU        *
        CONTROL    DS1,CLSOFF
*
        PRINTEXT   '@ END TEST PROGRAM@'
*
        PROGSTOP
*
*
ERR     EQU   *
        PRINTEXT   '@I/O ERROR - RC=   '
*
        PRINTNUM   DS1
*
        PRINTEXT   'TEST PROGRAM ENDING@'
*
        GOTO       ENDIT
*
*
BUFF    DATA       75C'TEST'
*
        ENDPROG
        END
```

EXAMPLE 16: PROCESSING STANDARD LABELS USING BLP

This example reads and processes the records of standard labels prior to reading and processing the data records in the tape data set. The tape is mounted on a tape drive whose configurated TAPEID is TAPE01. The tape drive has been assigned the attribute of BLP.

The first instruction reads the volume label (VOL1), whose length is 80 bytes, into a buffer labeled BUFFER, where it can be processed by your application program. The same buffer is used throughout the program. The second read instruction reads the first header label (HDR1), whose length is 80 bytes, into the buffer for processing by your application program. A CONTROL command (FSF) is then issued to space the tape past any additional header labels by searching for a tape mark. The program now reads data records from the tape, one record at a time, into the buffer for processing by your program. The data records are each 50 bytes long. When the last data record has been read and processed the 80 byte trailer record (EOF1) is read into the buffer and can be processed by your program.

If any errors are detected, while reading labels, the error routine named ERR1 is given control and the message LABEL ERROR - RC= is printed and the associated return code is printed to help you determine what type of error was encountered. If an error is detected during the reading of data records, the error routine named ERR2 is given control and the message READ ERROR - RC= is printed along with the return code which indicates the type of error encountered.

```
SLPROC  PROGRAM START,DS=((XYZ,TAPE01))
        START   EQU     *
*
*  PROCESS THE HEADER LABEL GROUP
*
         READ   DS1,BUFFER,1,80,ERROR=ERR1   Read the volume
*                                            1 label (VOL1)
*
*  PROCESS THE VOLUME 1 RECORD
*
         READ   DS1,BUFFER,1,80,ERROR=ERR1   Read the header
*                                            label (HDR1)
*
*  PROCESS THE HEADER 1 RECORD
*
         CONTROL  DS1,FSF               Space the tape past
*                                       any other label
*                                       records and the
*                                       tape mark
```

```
*
*  PROCESS THE DATA ON THE TAPE
*
LOOP     EQU    *
         READ   DS1,BUFFER,1,50,ERROR=ERR2,END=ALLDONE
*
*  PROCESS THE TAPE DATA RECORD JUST READ INTO BUFFER.
*  YOU MAY WISH TO:
*     PRINT IT
*     WRITE IT TO DISK OR DISKETTE
*     DISPLAY IT ON A TERMINAL
*     USE IT IN CALCULATIONS
*
         GOTO   LOOP                     Return to LOOP to
*                                        read the next data
*                                        record
ALLDONE  EQU    *
*
*  PROCESS THE TRAILER LABEL GROUP
*
         READ   DS1,BUFFER,1,80,ERROR=ERR1
*
*  PROCESS THE END OF FILE (EOF1) RECORD
*
ENDIT    EQU    *
         PROGSTOP
*
ERR1     EQU    *
         PRINTEXT  '@LABEL ERROR - RC= '
         PRINTNUM   DS1
         GOTO   ENDIT
ERR2     EQU    *
         PRINTEXT  '@READ ERROR - RC= '
         PRINTNUM   DS1
         QUESTION  'DO YOU WANT TO CONTINUE? ',
                    YES=LOOP,NO=ENDIT
*
BUFFER   DATA       40F'0'

         ENDPROG
         END
```

EXAMPLE 17: WRITE A DATA SET TO A SL TAPE THEN READ IT

This example uses a standard labeled (SL) tape to write a data set. The tape data set name is MYDATA and the volume serial number is 1004. The tape record must be created prior to the WRITE statement by moving a data record into BUFFER. The records are assumed to be 500 bytes long; longer records would be truncated to 500 bytes, shorter records would be padded to 500 bytes. After writing the data set, the tape is rewound. The tape data set is then reopened by a CALL to DSOPEN and the records are read back into storage at location BUFFER.

```
WRTAPE PROGRAM  START,DS=((MYDATA,1004))
START    EQU    *
*
         DO     100,TIMES     Write 100 records to tape
*
*   YOU MUST CREATE THE TAPE RECORD HERE; THE RECORD TO
*   BE WRITTEN TO TAPE MUST BE AT LOCATION BUFFER FOR
*   THIS EXAMPLE.
*
           WRITE  DS1,BUFFER,1,500,ERROR=ERR1
         ENDDO
DONE1    EQU    *
         CONTROL  DS1,CLSRU
*
*   SET THE DSOPEN ERROR EXITS
*
         MOVEA  $DSNFND,ERRDSN
         MOVEA  $DSBIODA,ERRIODA
         MOVEA  $DSBVOL,ERRVOL
         MOVEA  $DSIOERR,ERRIO
*
*   OPEN THE DATA SET
         CALL   DSOPEN,(DS1)           Reopen the data set
*                                      indicated in the
*                                      PROGRAM statement
*
*   READ AND PROCESS THE RECORDS JUST CREATED AND WRITTEN
*   TO THE TAPE DATA SET NAMED MYDATA
*
LOOP     EQU    *
         READ   DS1,BUFFER,1,500,ERROR=ERR2,END=DONE2,
                WAIT=YES
*
*   HERE THE RECORDS MUST BE MOVED OUT OF LOCATION BUFFER
*   BY YOUR PROGRAM, TO PREVENT THEM BEING OVER WRITTEN
*   BY THE NEXT RECORD FROM TAPE.
*
         GOTO   LOOP
DONE2    EQU    *
         CONTROL  DS1,CLSOFF
         PROGSTOP
```

```
ERR1      EQU     *
          PRINTEXT   '@WRITE ERROR - RC= '
          PRINTNUM    DS1
          QUESTION   'DO YOU WANT TO CONTINUE? ',
                      YES=START,NO=DONE1
*
ERR2      EQU     *
          PRINTEXT   '@READ ERROR - RC= '
          PRINTNUM    DS1
          QUESTION   '@DO YOU WANT TO CONTINUE? ',
                      YES=LOOP,NO=DONE2
*
BUFFER    DATA    250F'0'           Define a buffer of 500
                                    bytes and initialize
                                    it to zeros
*
*    DSOPEN ERROR EXITS, BUFFER AREA, AND COPY CODE
*
ERRDSN    EQU     *
          MOVEA   MSGX,MSG1
          GOTO    ERRMSG
*
ERRIODA   EQU     *
          MOVEA   MSGX,MSG2
          GOTO    ERRMSG
*
ERRVOL    EQU     *
          MOVEA   MSGX,MSG3
          GOTO    ERRMSG
*
ERRIO     EQU     *
          MOVEA   MSGX,MSG$
*
ERRMSG    EQU     *
          PRINTEXT   '@DSOPEN ERROR - '
          PRINTEXT   MSG1,P1=MSGX
          PRINTEXT   SKIP=1
          GOTO       DONE2
MSG1      TEXT       'DATA SET NOT FOUND'
MSG2      TEXT       'VOLUME NOT FOUND'
MSG3      TEXT       'I/O ERROR'
MSG4      TEXT       'DATA SET NOT FOUND'
          COPY       DSOPEN
          COPY       DSCBEQU
          COPY       DDBEQU
          COPY       PROGEQU
DISKBUFR DATA        128F'0'        Define a buffer area of
*                                   256 bytes and initialize
*                                   to zeros
*

          ENDPROG
          END
```

EXAMPLE 18: INITIALIZE AND WRITE A NL TAPE

This example uses the Utilities, Operator commands, and EDL instructions to initialize a tape and write a data set to the tape without using tape labels.

You must mount the tape on a drive defined for NL processing. If the drive is not defined for NL, then use $TAPEUT1 utility and the subcommand CT, to change the label processing attribute to NL. The procedure for preparing the tape for use follows and the bold type represents what you must enter from the keyboard:

```
$L $TAPEUT1                 (This loads the tape utility)

COMMAND (?)  IT             (This selects the initialize utility)

TAPE ADDR (1 - 2 HEX CHARS):  48      (Select the drive to
                                            be used)

NO LABEL 1600 BPI? Y        (Verifies the tape attributes)

TAPE INITIALIZED                  (Tape has been initialized)

COMMAND ?  EN              (This ends the tape utility session)


$VARYON 48                   (This will vary the tape online)
TAPE01 ONLINE              (The system responds with the tape
                           ID that was assigned during system
                           configuration)

$L PRGTAPE                     (System will load your program
                                  PRGTAPE and write the tape
                                  data set)
```

The program writes data to the tape to create the tape data set defined as MYDATA. It writes one record each time the DO loop is executed. The records are specified to be 50 bytes long. The data records are taken from a location labeled BUFFER. If a tape I/O error is detected during the writing of the data set, the program branches to label ERR1. In the error routine, ERR1, the return code indicating the type of error encountered is dispalyed and you are requested to respond whether you wish to resume the WRITE operation or not. If you reply YES on the keyboard, the DO loop will be resumed. If you reply NO, the program branches to the ending routine labeled ALLDONE.

```
PRGTAPE    PROGRAM   START,DS=((MYDATA,TAPE01))
START      EQU       *
           DO        100,TIMES
*
*  Create or build the tape record so that the data
*  you wish to write to tape is at location BUFFER.
*  For example you may:
*  - read from disk or diskette to BUFFER
*  - read from a terminal to BUFFER
*  - move records from a calculation in storage
*    to BUFFER
*
             WRITE     DS1,BUFFER,1,50,ERROR=ERR1
           ENDDO
*
ALLDONE    EQU       *
           CONTROL   DS1,CLSOFF
           PROGSTOP
*
ERR1       EQU       *
           PRINTEXT '@WRITE ERROR - RC= '
           PRINTNUM  DS1
*
           QUESTION '@DO YOU WISH TO RESUME?',YES=START,NO=ALLDONE
*
BUFFER     DATA       25F'0'        Creates the area from
                                    which the source data
                                    records will be written
*
           COPY       TDBEQU        Required for all
*                                   CONTROL requests
           ENDPROG
           END
```

EXAMPLE 19: READ THE THIRD FILE ON TAPE

This example shows the procedure for setting up an existing tape to read the third file whose data set name is MYDATA. The third file will be read one record at a time. The records are expected to be 50 bytes long. The records could be any length but the READ statement will only read 50 bytes and place them into location BUFFER. If the records in the third file are not 50 bytes in length longer records will be truncated to the right and shorter records will be padded on the right to fill the 50 word buffer.

When a tape mark is read, at the end of the third file, the tape will be close and placed offline by the CONTROL statement at label all done.

If a tape I/O error occurs while reading records from the file, the return code will be printed on the terminal and you will be prompted with a question. If you reply YES, the program will attempt to continue reading records from the third file. If you reply NO, the program will branch to label ALLDONE and the program will close the data set and place the tape offline.

The procedure for preparing the tape for use follows and the bold type represents what you must enter from the keyboard:

$VARYON 48,3	(This will vary the tape online)
TAPE01 ONLINE	(The system responds with the tape ID that was assigned during system configuration)
$L RDTHIRD	(System will load your program RDTHIRD and read the tape data set)

The EDL program follows:

```
RDTHIRD   PROGRAM   START,DC=((MYDATA,100104))
START     EQU       *
          READ      DS1,BUFFER,1,50,END=ALLDONE,ERROR=ERR1
*
*  Process the tape record. For example, you may:
*  - PRINT it
*  - WRITE it to disk, or diskette
*  - DISPLAY it on a terminal
*  - Use it in calculations
*
*  The record must be moved from BUFFER to prevent
*  the next record from overlaying it.
*
GOTO      START
*
ALLDONE   EQU       *
          CONTROL   DS1,CLSOFF
          PROGSTOP
*
ERR1      EQU       *
          PRINTEXT  '@READ ERROR - RC = '
          PRINTNUM  DS1
          QUESTION  '@ DO YOU WISH TO CONTINUE?',
                    YES=START,NO=ALLDONE
*
BUFFER    DATA      25F'0'
          COPY      TDBEQU
          ENDPROG
          END
```

EVENT DRIVEN LANGUAGE INSTRUCTIONS

The following syntax conventions are used for the Event Driven Language descriptions.

- Superscript 0 indicates indexable operand
- Brackets [ ] indicate optional operands
- Operands not enclosed in brackets are required
- Underscored items are default values
- The OR symbol | indicates mutually exclusive operands

Instruction / Operands

ADD
opnd1^0,opnd2^0[,count 1 - 32767][,RESULT=^{0}opnd1| variable][,PREC=S|D][,P1=,P2=,P3=]

ADDV
opnd1^0,opnd2^0,count 1 - 32767[,RESULT=^{0}opnd1| variable][PREC=S|D][,P1=,P2=,P3=]

AND
opnd1^0,opnd2^0[,count (1 - 32767,BYTE|WORD|DWORD)] [,RESULT=^{0}opnd1|var|vector][,P1=,P2=,P3=]

ATTACH
taskname[,priority 1 - 510|256][,CODE=code word| -1][,P1=,P2=,P3=]

ATTNLIST
(cc1,loc1[,...,ccn,locn])[,SCOPE=LOCAL|GLOBAL]

BSCCLOSE
bsciocb^0[,ERROR=label][,P1=,P2=]

BSCIOCB
lineaddr[,buffer1 addr,length1][,buffer2 addr, length2][,pollseq][,pollsize][,P1=,...,P7=]

BSCLINE
[ADDRESS=0 - FF|9][,TYPE=PT|SM|SA|MC|MT] [,RETRIES=6|value][,MC=NO|YES][,END=NO|YES]

BSCOPEN
bsciocb^0[,ERROR=label][,P1=,P2=]

BSCREAD
type C|D|E|I|P|Q|R|U,bsciocb^0[,ERROR=label] [,END=label][,TIMEOUT=YES|NO][,P1=,P2=,P3=]

BSCWRITE	type C|CV|CVX|CX,CXB|D|E|EX|I|IV|IVX|IX|IXB|Q|N|U|UX,bsciocb°[,ERROR=label][,END=label][,CHECK=YES|NO][,P1=,P2=,P3=]
BUFFER	count 1 - 32767[,WORD|BYTE][,INDEX=name]
CALL	name[,par1,...,par5][,P1=,...,P6=]
CALLFORT	name[,(a1,a2,...,an,)][,P=(p1,p2,...pn)]
CONCAT	text1,text2[,RESET][,REPEAT=1 - 32767][,P1=,P2=]
CONTROL	DSx,BSF|FSF|BSR|FSR|WTM|REW|ROFF|OFF|CLSRU|CLSOFF[,count° 1 - 32767][,END=label][,ERROR=label][,WAIT=YES|NO][,P3=]
CONVTB	opnd1°,opnd2°[,PREC=S|D|F|L][,FORMAT=(w,d,t)|(6,0,I)][,P1=,P2=]
CONVTD	opnd1°,opnd2°[,PREC=,S|D|F|L][,FORMAT=(w,d,t)|(6,0,I)][,P1=,P2=]
COPY	symbol
CSECT	(label required)
DATA	[dup] type C|X|B|F|H|D|E|L|A value
DC	[dup] type C|X|B|F|H|D|E|L|A value
DCB	[,PCI=NO|YES][,IOTYPE=OUTPUT|INPUT][,XD=NO|YES][,SE=NO|YES][,DEVMOD=hex value][,DVPARM1=value|+label][,DVPARM2=value|+label][,DVPARM3=value|+label][,DVPARM4=value|label][,CHAINAD=label][,COUNT=0 - 332767|+label][,DATADDR=label] (label required)
DEFINEQ	COUNT=value[,SIZE=value] (label required)
DEQ	resource°[,code value|-1][,P1=,P2=]
DEQT	none
DETACH	[code value|-1][,P1=]
DIVIDE	opnd1°,opnd2°[,count value|1][,RESULT=°label|opnd1][,PREC=S|SSD|D|DD|DSS][,P1=,P2=,P3=]
DO	count 0 - 32767°[,TIMES][,INDEX=label][,P1=]|UNTIL,statement|WHILE,statement

DSCB	DS#=name,DSNAME=name[,VOLSER=name|null][,DSLEN=0 - maximum-direct-access-space-value]
ECB	[code value|-1] (label required)
EJECT	none (label not allowed)
ELSE	none
END	none (label not allowed)
ENDATTN	none
ENDDO	none
ENDIF	none
ENDPROG	none (label not allowed)
ENDTASK	[-1|posting code value][,P1=]
ENQ	resource^{0}[,BUSY=busy addr][,P1=]
ENQT	[name][,BUSY=][,P1=]
ENTRY	symbol1[,...,symboln]
EOR	opnd1^{0},opnd2^{0}[,(count 1 - 32767,BYTE|WORD|DWORD)] [,RESULT=^{0}opnd1|variable][,P1=,P2=,P3=]
EQU	value (label required)
ERASE	[count^{0}=maximum|value][,MODE=FIELD|LINE|SCREEN] [,TYPE=DATA|ALL][,SKIP=^{0}0 - pagesize][,LINE=^{0}0 - pagesize\|current line][,SPACES=^{0}0 - line spaces
EXIO	idcbaddr^{0}[,ERROR=label][,P1=]
EXOPEN	devaddr,listaddr^{0}[,ERROR=label][,P1=,P2=]
EXTRN	symbol1[,...,symboln] (label not allowed)
FADD	opnd1^{0},opnd2^{0}[,RESULT=^{0}opnd1|variable] [,PREC=FFF|DSD|SSD|SSS|DSS][,P1=,P2=,P3=]
FDIVD	opnd1^{0},opnd2^{0}[,RESULT=^{0}opnd1|variable] [,PREC=FFF|DSD|SSD|SSS|DSS][,P1=,P2=,P3=]
FIND	character,string^{0},length^{0},where^{0},notfound [,DIR=FORWARD|REVERSE][,P1=,P2=,P3=,P4=,P5=]

FINDNOT	character,string⁰,length⁰,where⁰,notfound [,DIR=FORWARD|REVERSE][,P1=,P2=,P3=,P4=,P5=]
FIRSTQ	qname⁰,loc⁰[,EMPTY=][,P1=,P2=]
FMULT	opnd1⁰,opnd2⁰[,RESULT=⁰opnd1|variable] [,PREC=FFF|DSD|SSD|SSS|DSS][,P1=,P2=,P3=]
FORMAT	(list),[GET|PUT|BOTH]
FPCONV	opnd1⁰,opnd2⁰[,COUNT=1 - 32767] [,PREC=FS|*|LD|DL|SF|FS][,P1=,P2=,P3=]
FSUB	opnd1⁰,opnd2⁰[,RESULT=⁰opnd1|variable] [,PREC=FFF|*|any combination][,P1=,P2=,P3=]
GETEDIT	text,(list)format|(format list)[,ERROR=label] [,ACTION=IO|STG][,SCAN=FIXED|FREE] [,SKIP=0 - pagesize][,LINE=0 - pagesize] [,SPACES=0 - linesize][,PROTECT=NO|YES]
GETTIME	loc⁰[,DATE=NO|YES][,P1=]
GETVALUE	loc⁰[,pmsg⁰|label][,count 1 - 32767|(count value, BYTE|WORD|DWORD)] [,MODE=DEC|HEX][,PROMPT=UNCOND|COND] [,FORMAT=(6,0,I)|(w,d,f)][,TYPE=S|D|F|L] [,SKIP=⁰,0 - pagesize][,LINE=⁰current line| 0 - pagesize][,SPACES=⁰0 - linesize][,P1=,P2=,P3=]
GIN	x,y[,char,][P1=,P2=,P3=]
GOTO	loc[,P1=]
GOTO	(loc0[,loc1,loc2,...,loc49])[,index⁰][,P1=,P2=]
IDCB	COMMAND=READ|READ1|READID|RSTATUS|WRITE|WRITE1| PREPARE|CONTROL|RESET|START|SCSS,ADDRESS=label[, DCB=dcb label][,DATA=addr][,MOD4=modifier][, LEVEL=0 - 3|1][,IBIT=ON|OFF] (label required)
IF	statement[,GOTO,loc]
INTIME	reltime,loc[,INDEX][,P2=]
IOCB	[name][,PAGSIZE=][,TOPM=0 - pagesize-1] [,BOTM=0 - pagesize-1][,LEFTM=0 - linesize-1] [,RIGHTM=0 - linesize-1][,SCREEN=ROLL|STATIC] ,[NHIST=0 - pagesize-2] [,OVFLINE=NO|YES][BUFFER=RITHTM+1 - 32767]

IODEF	PIx,ADDRESS=00 - FF[,TYPE=GROUP]|[TYPE=BIT,] BIT=0 - 15[,SPECPI=]
IODEF	DOx,TYPE=GROUP|SUBGROUP,BITS=(u 0 - 15,v 1 - 16-u)| EXTSYNCADDRESS=00 - FF
IODEF	DIx,TYPE=GROUP|SUBGROUP,BITS=(u 0 - 15,v 1 - 16-u)| EXTSYNCADDRESS=00 - FF
IODEF	AOx,ADDRESS=00 - FF,POINT=0 - 1
IODEF	AIx,ADDRESS=00 - FF,POINT=0 - 7 for relay, 0 - 15 for ss,RANGE=5V|500MV|200MV|100MV| 50MV|20MV|10MV,ZCOR=NO|YES
IOR	opnd1°,opnd2°[,count 1 - 32767,BYTE|WORD|DWORD] [,RESULT=°opnd1| variable][,P1=,P2=,P3=]
LASTQ	qname°,loc°[,EMPTY=label][,P1=,P2=]
LOAD	prog name[[,parm name][,DS=(dsname1,...,dsname9)] [,EVENT=event name][,LOGMSG=YES|NO][,PART=1 - 8] [,ERROR=label][,STG=0 - 65535][,P2=]
LOAD	PGMx[[,parm name][,DS=(DSx,...)][, EVENT=event name][,LOGMSG=YES|NO][, ERROR=label][,P2=]
MOVE	opnd1°,opnd2°[,count 1 - 32767,BYTE|WORD|DWORD] [,FKEY=][,TKEY=][,P1=,P2=,P3=]
MOVEA	opnd1°,opnd2[,P1=,P2=]
MULTIPLY	opnd1°,opnd2°[,count 1 - 32767] [,RESULT=°opnd1|variable,][,PREC=S|D] [,P1=,P2=,P3=]
NEXTQ	qname°,loc°[,FULL=][,P1=,P2=]
NOTE	DSx,loc°[,P2=]
PLOTCB	8 data statements with explicit values
PLOTGIN	x,y[,char],pcb[,P1=,P2=,P3=,P4=]
POINT	DSx,relrecno°[,P2=]
POST	event°[,code -1 - FF[,P1=,P2=]
PRINDATE	none

PRINT [ON|OFF][,GEN|NOGEN][,DATA|NODATA]
(label not allowed)

PRINTEXT msg°|[,SKIP=°0 - pagesize]| [,LINE=°
current line]|[SPACES=°0 - line
length-1][,XLATE=°YES|NO]
[,MODE=][,PROTECT=NO|YES][,P1=]

PRINTIME none

PRINTNUM loc°[,count 1 - 32767,WORD|DWORD] [,nline=1
- line length-1] [nspace=1line length-2]
[,MODE=DEC|HEX][FORMAT=(6,0,I)|(w,d,f)]
[,TYPE=S|D|F|L]
[,SKIP=°0|pagesize-1][,LINE=°current line|
pagesize-1][SPACES=°0 linesize-1]
PROTECT=NO|YES][,P1=,..,P4=]

PROGRAM start label[,priority=150|1 -
510][,EVENT=name]
[,DS=(dsname1,..,dsname9)][,PARM=0 - 368]
[,PGMS=(pgmname1,..,pgmname9)]
[TERMERR=label,][,FLOAT=NO|YES][MAIN=YES|NO]
[,ERRXIT=label][STG=0 -
65535][,WXTRN=YES|NO] (taskname required
for label)

PROGSTOP [code -1 - FF][,LOGMSG=YES|NO][,P1=]

PUTEDIT text,(list),format|(format
list),[ERROR=label] [,ACTION=IO|G][,SKIP=0
- pagesize-1] [,LINE=1 -
pagesize-1][,SPACES=0 - linelength-1]
[,PROTECT=NO|YES]

QCB [code -1 - 99] (label required)

QUESTION pmsg° YES=label|NO=label[,SKIP=°0 -
pagesize-1] [,LINE=°1 -
pagesize-1][,SPACES=°1 -
linelength-1][,P1=]

RDCURSOR line°name,indent°name

READ DSx,loc°[,count° 1 - 32767° [,relrecno°0 -
max records-1|blksize° 256|[18 - 32767]]
[,END=label][,ERROR=label][,WAIT=YES|NO[,P2=,P3=,P4=]

READTEXT loc°[,pmsg°][,PROMPT=UNCOND|COND]
[,ECHO=NO][,TYPE=DATA|MODDATA|ALL|MODALL]
[,MODE=WORD|LINE][,XLATE=NO][,SKIP=°0 -
pagesize-1] [,LINE°=current line - bottom
line-1] [,SPACES=°0 - line length-1]

RESET	event^0 or for PI 1 - 99[,P1=]
RETURN	none
SBIO	AIx[[,loc^0],op3 label|INDEX][,SEQ=NO|YES] [,P1=,P2=,P3=]
SBIO	AOx[,loc^0][INDEX][,EOB=label][,P1=,P2=]
SBIO	DIx[,loc^0][INDEX][,EOB=label][ERROR=label] [,P1=,P2=]
SBIO	DIx[[,loc^0][,BITS=(u 0 - 15,v 0 n)][, ERROR=label][,P1=,P2=]
SBIO	DOx[[,loc^0][,BITS=(u 0 - 15,v 0 - n)][, ERROR=label][LSB=0 - 15][,P1=,P2=]
SBIO	DOx[,loc^0][,op3 label|INDEX][, ERROR=label][,P1=,P2=,P3=,]
SBIO	DOx,(PULSE,ON|OFF)
SCREEN	text,x,y[,CONCAT=NO|YES][,ENHGR=NO|YES] [,P1=,P2=,P3=]
SHIFTL	opnd1^0,opnd2^0[,count 1 - 32767[,BYTE|WORD| DWORD]][,RESULT=^{0}opnd1|label][,P1=,P2=,P3=]
SHIFTR	opnd1^0,opnd2^0[,count 1 - 32767[,BYTE|WORD| DWORD]][,RESULT=^{0}opnd1|label][,P1=,P2=,P3=]
SPACE	[value 1 - pagesize-1] (label not allowed)
SPECPIRT	
SQRT	rsq^0,root,rem[,P1=,P2=,P3=]
STATUS	index,key[,length 0 - 256][,P1=,P2=,P3=] (label required)
STIMER	count^0 1 - 32767[,WAIT][,P1=]
SUBROUT	name[,par1,...,par5]
SUBTRACT	opnd1^0,opnd2^0[,count 1 - 32767][RESULT=^{0}opnd1] [PREC=S|D][,P1=,P2=,P3=]
TASK	start[,priority 150|1 - 510][EVENT=name] [,TERMERR=label][FLOAT=YES][ERRXIT=label] (taskname required for label)

TERMCTRL	function BLANK|DISPLAY|TONE|BLINK|UNBLINK|LOCK|UNLOCK|PF,code|SET,ATTN=YES|NO|LET,LPI=6|8|PUTSTORE|GETSTORE [,op1° addr][,op2° addr][,TYPE=1|2|4|5|6|7]
TEXT	'message'|LENGTH=1 - 254[,CODE=E|A]
TITLE	message (label not allowed)
TP	CLOSE[,ERROR=]
TP	FETCH,stloc°[,length° 0 - 256][,ERROR=label] [,P2=,P3=]
TP	OPENIN,dsnloc°[,ERROR=label][,P2=]
TP	OPENOUT,dsnloc°[,ERROR=label][,P2=]
TP	READ,buffer°[,count° 1 - 32767][,END=label] [,ERROR=label][,P2=,P3=]
TP	RELEASE,stloc°[,length° 0 - 256][,ERROR=label] [,P2=,P3=]
TP	SET,stloc°[,length° 0 - 256][,ERROR=label] [,P2=,P3=]
TP	SUBMIT,dsnloc°[,ERROR=label][,P2=]
TP	TIMEDATE,loc°[,ERROR=label][,P2=]
TP	WRITE,buffer°[,count° 1 - 32767] [,relrecno° 0 - 32767|blksize° 256|[18 - 32767] [,END=label][,ERROR=label][P2=,P3=]
USER	name[,PARM=(parm1,..,parmn)] [,P=(name1,..,namen)]
WAIT	event°[,RESET][,P1=]
WHERES	progname,address[,KEY][,P1=,P2=,P3=]
WRITE	DSx,loc°[,count° 1 - 32767] [,relrecno°0 - max records-1|blksize° 256|[18 - 32766]] [,END=label] [,ERROR=label] [WAIT=YES|NO][,P2=,P3=,P4=]
WXTRN	symbol1[,symbol2,...,symboln]
XYPLOT	x,y,pcb,n[P1=,P2=,P3=,P4=]
YTPLOT	y,x1,pcb,n,inc[,P1=,P2=,P3=,P4=,P5=]

INDEXED ACCESS METHOD

Instruction	Operands
CALL	IAM,(DEFINE)
CALL	IAM,(DELETE),iacb,(key)
CALL	IAM,(DISCONN),iacb
CALL	IAM,(ENDSEQ),iacb
CALL	IAM,(EXTRACT),iacb,(buff-addr) [,(size FULL|byte-value)]
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)

EVENT DRIVEN EXECUTIVE LIBRARY SUMMARY

The library summary is a guide to the Event Driven Executive library. By briefly listing the content of each book and providing a suggested reading sequence for the library, it should assist you in using the library as a whole as well as direct you to the individual books you require.

Event Driven Executive Library

The IBM Series/1 Event Driven Executive library materials consist of five full-sized books, a quick reference pocket book, and a set of tabs:

- IBM Series/1 Event Driven Executive System Guide (or System Guide), SC34-0312
- IBM Series/1 Event Driven Executive Utilities, Operator Commands, Program Preparation, Messages and Codes (or Utilities), SC34-0313
- IBM Series/1 Event Driven Executive Language Reference (or Language Reference), SC34-0314
- IBM Series/1 Event Driven Executive Communications and Terminal Application Guide (or Communications Guide), SC34-0316
- IBM Series/1 Event Driven Executive Internal Design (or Internal Design), LY34-0168
- IBM Series/1 Event Driven Executive Multiple Terminal Manager Internal Design (or Multiple Terminal Manager Internal Design), LY34-0190
- IBM Series/1 Event Driven Executive Indexed Access Method Internal Design (or Indexed Access Method Internal Design), LY34-0189
- IBM Series/1 Event Driven Executive Reference Summary (or Reference Summary), SX34-0101
- IBM Series/1 Event Driven Executive Tabs (or Tabs), SX34-0030

Summary of Library

System Guide

The System Guide introduces the concepts and capabilities of the Event Driven Executive system. It discusses multi-tasking, program and task structure, program overlays, storage management, and data management.

Planning aids include hardware and software requirements, along with guidelines for storage estimating.

The System Guide also presents step-by-step procedures for generating a supervisor tailored to your Series/1 hardware configuration and software needs.

The description of the Indexed Access Method contains the information on how to write applications that use indexed data sets.

The description of the session manager includes a procedure for modifying the session manager to include application programs in the primary option menu so that you can execute them under the session manager. You can also add a procedure to compile, link, and update programs.

Information is also provided concerning partitioned data sets, tape data organization, diagnostic aids, inter-program communication, logical screens, and dynamic data set allocation.

Utilities

Utilities describes:

- Event Driven Executive utility programs
- Operator commands
- Procedures to prepare and execute system and application programs
- The session manager -- a menu-driven interface program that will invoke the programs required for program development
- Messages and codes issued by the Event Driven Executive system

The operator commands, program preparation facilities, and session manager are grouped by function and discussions include detailed syntax and explanations. The utilities are presented in alphabetical order.

Language Reference

The Language Reference familiarizes you with the Event Driven Language by first grouping the instructions into functional categories. Then the instructions are listed alphabetically, with complete syntax and an explanation of each operand.

The final section of the Language Reference contains examples of using the Event Driven language for applications such as:

- Program loading
- User exit routine
- Graphics
- I/O level control program
- Indexing and hardware register usage

Communications Guide

The Communications Guide introduces the Event Driven Executive communications support -- binary synchronous communications, asynchronous communications, and the Host Communications Facility.

The Communications Guide contains coding details for all utilities and Event Driven language instructions needed for communications support and advanced terminal applications.

Internal Design

Internal Design describes the internal logic flow and specifications of the Event Driven Executive system so that you can understand how the system interfaces with application programs. It familiarizes you with the design and implementation by describing the purpose, function, and operation of the various Event Driven Executive system programs.

Multiple Terminal Manager Internal Design and Indexed Access Method Internal Design describe the internal logic flow and specifications of these programs.

Unlike the other manuals in the library, the Internal Design books contain material that is the licensed property of IBM and they are available only to licensed users of the Event Driven Executive system.

Reference Summary

The Reference Summary is a pocket-sized booklet to be used for quick reference. It lists the Event Driven language instructions with their syntax, the utility and program preparation commands, and the completion codes.

Tabs

The tabs package must be ordered separately. The package contains 33 index tabs by subject, with additional blank tabs. These extended tabular pages can be inserted at the front of various sections of the library. The tabs are color coded according to the major library topics.

Reading Sequence

All readers of the Event Driven Executive library should begin with the first three chapters of the System Guide ("Introduction," "The Supervisor and Emulator," and "Data Management") for an overview of the Event Driven Executive concepts and facilities.

Readers responsible for installing and preparing the system should then continue in the System Guide with "System Configuration" and "System Generation."

All readers should review the Utilities "Introduction" to become familiar with the utility functions available for the Event Driven Executive system. Then you can read more specific sections for particular utilities, operator commands, and program preparation facilities.

After you have a basic understanding of the Event Driven Executive system and how you can best use the system for your applications, you should read the Language Reference "Introduction." This will familiarize you with the potential of the Event Driven Language and prepare you to start coding application programs.

If you have communications support for your Event Driven Executive system, you should read the Communications Guide, which is an extension of the System Guide, Utilities, and the Language Reference.

After you know the functions of the various Event Driven Language instructions, utilities, and program preparation facilities, you may wish to refer only to the Reference Summary for correct syntax while coding your applications.

Only readers responsible for the support or modification of the Event Driven Executive system need to read Internal Design.

OTHER EVENT DRIVEN EXECUTIVE PROGRAMMING PUBLICATIONS

- IBM Series/1 Event Driven Executive FORTRAN IV User's Guide, SC34-0315.
- IBM Series/1 Event Driven Executive PL/I Language Reference, GC34-0147.
- IBM Series/1 Event Driven Executive PL/I User's Guide, GC34-0148.
- IBM Series/1 Event Driven Executive COBOL Programmer's Guide, SL23-0014.
- IBM Series/1 Event Driven Executive Sort/Merge Programmer's Guide, SL23-0016
- IBM Series/1 Event Driven Executive Macro Assembler Reference,GC34-0317.
- IBM Series/1 Event Driven Executive Study Guide, SR30-0436.

OTHER SERIES/1 PROGRAMMING PUBLICATIONS

- IBM Series/1 Programming System Summary, GC34-0285.
- IBM Series/1 COBOL Language Reference, GC34-0234.

- IBM Series/1 FORTRAN IV Language Reference, GC34-0133.
- IBM Series/1 Host Communications Facility Program Description Manual, SH20-1819.
- IBM Series/1 Mathematical and Functional Subroutine Library User's Guide, SC34-0139.
- IBM Series/1 Macro Assembler Reference Summary, SX34-0128
- IBM Series/1 Data Collection Interactive Programming RPQ P82600 User's Guide, SC34-1654.

OTHER PROGRAMMING PUBLICATIONS

- IBM Data Processing Glossary, GC20-1699.
- IBM Series/1 Graphic Bibliography, GA34-0055.
- IBM OS/VS Basic Telecommunications Access Method (BTAM), GC27-6980.
- General Information — Binary Synchronous Communications, GA27-3004.
- IBM System/370 Program Preparation Facility, SB30-1072.

SERIES/1 SYSTEM LIBRARY PUBLICATIONS

- IBM Series/1 4952 Processor and Processor Features Description, GA34-0084.
- IBM Series/1 4953 Processor and Processor Features Description, GA34-0022.
- IBM Series/1 4955 Processor and Processor Features Description, GA34-0021.
- IBM Series/1 Communications Features Description, GA34 -0028.
- IBM Series/1 3101 Display Terminal Description, GA34-2034.
- IBM Series/1 4962 Disk Storage Unit and 4964 Diskette Unit Description, GA34-0024.

- IBM Series/1 4963 Disk Subsystem Description, GA34-0051.
- IBM Series/1 4966 Diskette Magazine Unit Description, GA34-0052.
- IBM Series/1 4969 Magnetic Tape Subsystem Description, GA34-0087.
- IBM Series/1 4973 Line Printer Description, GA34-0044.
- IBM Series/1 4974 Printer Description, GA34-0025.
- IBM Series/1 4978-1 Display Station (RPQ D02055) and Attachment (RPQ D02038) General Information, GA34-1550
- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02056) General Information, GA34-1551
- IBM Series/1 4978-1 Display Station, Keyboard (RPQ D02057) General Information, GA34-1552
- IBM Series/1 4978-1 Display Station Keyboards (RPQ D02064 and D02065) General Information, GA34-1553
- IBM Series/1 4979 Display Station Description, GA34-0026
- IBM Series/1 4982 Sensor Input/Output Unit Description, GA34-0027
- IBM Series/1 Data Collection Interactive RPQs D02312, D02313, and D02314 Custom Feature, GA34-1567

This glossary contains terms that are used in the Series/1 Event Driven Executive software publications. All software and hardware terms are Series/1 oriented. This glossary defines terms used in this library and serves as a supplement to the IBM Data Processing Glossary (GC20-1699).

$SYSLOGA. The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG. The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR. The name of the system printer.

ACCA. See asynchronous communications control adapter.

address key. Identifies a set of Series/1 segmentation registers and represents an address space. It is one less than the partition number.

address space. The logical storage identified by an address key. An address space is the storage for a partition.

application program manager. The component of the Multiple Terminal Manager that provides the program management facilities required to process user requests. It controls the contents of a program area and the execution of programs within the area.

application program stub. A collection of subroutines that are appended to a program by the linkage editor to provide the link from the application program to the Multiple Terminal Manager facilities.

asynchronous communications control adapter. An ASCII terminal attached via #1610, #2091 with #2092, or #2095 with #2096 adapters.

attention list. A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup. A copy of data to be used in the event the original data is lost or damaged.

base records. Records that have been placed into an indexed data set while in load mode.

basic exchange format. A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB). A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

block. (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

BSCDDB. See binary synchronous device data block.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing. Access of a tape without any label processing support.

CCB. See terminal control block.

character image. An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table. An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

cluster. In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction). A character used with ACCA terminal to indicate a reverse in the direction of data movement.

command. A character string from a source external to the system that represents a request for action by the system.

common area. A user-defined data area that is mapped into every partition at the same address. It can be used to contain control blocks or data that will be accessed by more than one program.

completion code. An indicator that reflects the status of the execution of a program. The completion code is displayed or printed on the program's output device.

conversion. See update.

cross partition service. A function that accesses data in two partitions.

data block. In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data set. A group of contiguous records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB). A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down. An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE. See directory control entry.

DDB. See disk data block.

direct access. (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation.

directory. A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free spaces.

directory control entry (DCE). The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME). A 32-byte directory entry describing an allocated data set.

disk data block (DDB). A control block that describes a direct access volume.

display station. An IBM 4978 or 4979 display terminal or similar terminal with a keyboard and a video display.

DME. See directory member entry.

DSCB. See data set control block.

dynamic storage. An increment of storage that is appended to a program when it is loaded.

end-of-data indicator. A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB. See event control block.

EDL. See Event Driven Language.

emulator. The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT). A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

event control block (ECB). A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

event driven language (EDL). The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output). An EDL facility that provides user controlled access to Series/1 input/output devices.

external label. A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN). The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA. See file control area.

FCB. See file control block.

file control area (FCA). A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB). In an indexed data set, the first block of the data set. It contains descriptive information about the data contained in the data set.

file manager. A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

formatted screen image. A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool. In an indexed data set, a group of blocks that can be used as either a data block or an index block. These differ from other free blocks in that these are not initially assigned to specific logical positions in the data set.

free space. In the Indexed Access Method, record spaces or blocks that do not currently contain data, and are available for use.

free space entry (FSE). A 4-byte directory entry defining an area of free space within a volume.

FSE. See free space entry.

hardware timer. The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card or the native timer (4952 only). Only one or the other is supported by the Event Driven Executive.

host assembler. The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system. Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB. See indexed access control block.

IAR. See instruction address register.

ICB. See indexed access control block.

IIB. See interrupt information byte.

image store. The area in a 4978 that contains the character image table.

index. In the Indexed Access Method, an ordered collection of pairs, each consisting of a key and a pointer, used to sequence and locate the records in an Indexed Access Method data set.

index block. In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB). The control block that relates an application program to an indexed data set.

indexed access method. An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set. A data set specifically created, formatted and used by the Indexed Access Method. An indexed data set may also be called an indexed file.

indexed file. Synonym for indexed data set.

index entry. In an indexed file, a key-pointer pair, where the pointer is be used to locate a lower-level index block or a data block.

index register (#1, #2). Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB). A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer.

instruction address register (IAR). The pointer that identifies the instruction currently being executed. The Series/1 maintains a hardware IAR to determine the Series/1 assembler instruction being executed. It is located in the level status block (LSB).

interactive. The mode in which a program conducts a continuous dialogue between the user and the system.

internal label. An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB). In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

job. A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement. A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor. The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

key. In the Indexed Access Method, one or more consecutive characters in a data record, used to identify the record and establish its order with respect to other records. See also key field.

key field. A field, located in the same position in each record of an Indexed Access Method data set, whose content is used for the key of a record.

level status block (LSB). A Series/1 hardware data area that contains processor status.

library. A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line. A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit. The process of resolving symbols in one or more object modules to produce another single module that is the input to the update process.

load mode. In the Indexed Access Method, the mode in which records are initially placed in an indexed file.

load module. A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point. A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock. In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

LSB. See level status block.

member. A term used to identify a named portion of a partitioned data set (PDS). Sometimes member is also used as a synonym for a data set. See data set.

menu. A formatted screen image containing a list of options. The user selects an option to invoke a program.

menu-driven. The mode of processing in which input consists of the responses to prompting from an option menu.

multifile volume. A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager. An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file. A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

non-labeled tapes. Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character. A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu. A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay. The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program.

overlay area. A storage area within a program reserved for overlay programs specified in the PROGRAM statement.

parameter selection menu. A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition. A contiguous fixed-sized area of storage. Each partition is a separate address space.

physical timer. Synonym for hardware timer.

prefind. To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary-level index block. In an indexed data set, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu. The program selection screen displayed by the Multiple Terminal Manager.

primary option menu. The first full screen display provided by the Session Manager.

primary task. The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority. A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode. In the Indexed Access Method, the mode in which records may be retrieved, updated, inserted or deleted.

processor status word (PSW). A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program. A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header. The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager. A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field. On a display device, a field in which the operator cannot enter, modify, or erase data from the keyboard. It can contain text that the user can read.

PSW. See processor status word.

QCB. See queue control block.

QD. See queue descriptor.

QE. See queue element.

queue control block (QCB). A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD). A control block describing a queue built by the DEFINEQ instruction.

queue element (QE). An entry in the queue defined by the queue descriptor.

record. (1) The smallest unit of direct access storage that can be accessed by an application program on a disk or diskette using READ and WRITE. Records are 256 bytes in length. (2) In the Indexed Access Method, the logical unit that is transferred between $IAM and the user's buffer. The length of the buffer is defined by the user.

recovery. The use of backup data to recreate data that has been lost or damaged.

reflective marker. A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative record number. An integer value identifying the position of a record in a data set relative to the beginning of the data set. The first record of a data set is record one, the second is record two, the third is record three.

reorganize. For an indexed data set, the copying of the data to a new indexed data set in a manner that rearranges the data for more optimum processing and free space distribution.

return code. An indicator that reflects the results of the execution of an instruction or subroutine. The return code is placed in the task code word (at the beginning of the task control block).

roll screen. A display screen on which data is displayed 24 lines at a time or data is entered line by line, beginning with line 0 at the top of the screen and continuing through line 23 at the bottom of the screen. When a roll screen device's screen is full (all 24 lines used), an attempt to display the next line results in removal of the old screen (screen is erased) and the new line on line 0 is displayed at the top of the screen.

SBIOCB. See sensor based I/O control block.

second-level index block. In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary option menu. In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task. Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

sector. The smallest addressable unit of storage on a disk or diskette. A sector on a 4962 or 4963 disk is equivalent to an Event Driven Executive record. On a 4964 or 4966 diskette, two sectors are equivalent to an Event Driven Executive record.

sensor based I/O control block (SBIOCB). A control block containing information related to sensor I/O operations.

sequential access. The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR). A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

session manager. A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource. A resource that can be used by more than one task at the same time.

shut down. See data set shut down.

source module/program. A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

standard labels. Fixed length 80-character records on tape containing specific fields of information (a volume label identifying the tape volume, a header label preceding the data records, and a trailer label following the data records).

static screen. A display screen formatted with predetermined protected and unprotected areas. Areas defined as operator prompts or input field names are protected to prevent accidental overlay by input data. Areas defined as input areas are not protected and are usually filled in by an operator. The entire screen is treated as a page of information.

subroutine. A sequence of instructions that may be accessed from one or more points in a program.

supervisor. The component of the Event Driven Executive capable of controlling execution of both system and application programs.

system configuration. The process of defining devices and features attached to the Series/1.

SYSGEN. See system generation.

system generation. The processing of user selected options to create a supervisor tailored to the needs of a specific Series/1 configuration.

system partition. The partition that contains the supervisor (partition number 1, address space 0).

tapemark. A control character recorded on tape used to separate files.

task. The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task are secondary tasks.

task code word. The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB). A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor. The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB. See task control block.

terminal. A display station, teletypewriter or printer.

terminal control block (CCB). A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB). A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager. The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server. A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

trace range. A specified number of instruction addresses within which the flow of execution can be traced.

transaction oriented applications. Program execution driven by operator actions, such as responses to prompts from the system. Specifically, applications executed under control of the Multiple Terminal Manager.

transaction program. See transaction-oriented applications.

transaction selection menu. A Multiple Terminal Manager display screen (menu) offering the user a choice of functions, such as reading from a data file, displaying data on a terminal, or waiting for a response. Based upon the choice of option, the application program performs the requested processing operation.

unprotected field. On a display device, a field in which the user can enter, modify, or erase data using the keyboard. Unprotected fields on a static screen are defined by the null character.

update. (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit. (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a user written routine can be given control.

vary offline. (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is not available for use by the system; however, it will still be available for executing I/O at the basic access level (EXIO).

vary online. To restore a device to a state where it is available for use by the system.

volume. A disk or diskette subdivision defined during system configuration. A volume may contain up to 32,767 records. As many volumes may be defined for a disk as will physically fit. A diskette is limited to one volume.

volume label. A label that uniquely identifies a single unit of storage media.

This index is common to the Event Driven Executive library. The index includes entries from the seven publications listed below. (The Glossary is not indexed.) Each publication has a copy of the index, which provides a cross-reference between the publications.

Each page number entry contains a single letter prefix which identifies the publication where the listed subject can be found. The letter prefixes have the following meanings:

- C = Communications and Terminal Application Guide
- I = Internal Design
- L = Language Reference
- S = System Guide
- U = Utilities, Operator Commands, Program Preparation, Messages and Codes
- M = Multiple Terminal Manager Internal Design
- A = Indexed Access Method Internal Design

Special Characters

A

B

C

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

U

V

W

XYZ

Numeric Subjects

READER'S COMMENT FORM

SC34-0314-2

IBM Series/1 Event Driven Executive
Language Reference

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

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

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

Cut or Fold Along Line

Reader's Comment Form

Cut Along Line

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

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

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

IBM®

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

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

SC34-0314-2
Printed in U.S.A.

READER'S COMMENT FORM

SC34-0314-2

IBM Series/1 Event Driven Executive
Language Reference

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or the IBM branch office serving your locality.

Corrections or clarifications needed:

Page Comment

Cut or Fold Along Line

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

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

Reader's Comment Form

Cut Along Line

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

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

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

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

IBM®

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

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

SC34-0314-2
Printed in U.S.A.

International Business Machines Corporation

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

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

SC34-0314-2
Printed in U.S.A.