IBM

Series/1

SC34-0643-0

Event Driven Executive Language Reference

Version 5.0

Library Guide and Common Index SC34-0645	Installation and System Generation Guide SC34-0646	Operator Commands and Utilities Reference SC34-0644
Language Reference SC34-0643	Communications Guide SC34-0638	Messages and Codes SC34-0636
Operation Guide SC34-0642	Event Driven Language Programming Guide SC34-0637	Reference Cards SBOF-1625
Problem Determination Guide SC34-0639	Customization Guide SC34-0635	Internal Design LY34-0354

IBM

Series/1

SC34-0643-0

Event Driven Executive Language Reference

Version 5.0

Library Guide and Common Index SC34-0645	Installation and System Generation Guide SC34-0646	Operator Commands and Utilities Reference SC34-0644
Language Reference SC34-0643	Communications Guide SC34-0638	Messages and Codes SC34-0636
Operation Guide SC34-0642	Event Driven Language Programming Guide SC34-0637	Reference Cards SBOF-1625
Problem Determination Guide SC34-0639	Customization Guide SC34-0635	Internal Design LY34-0354

First Edition (December 1984)

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

Changes are made periodically to the information herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

This material may contain reference to, or information about, IBM products (machines and programs), programming, or services that 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 readers' comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Information Development, 3406, P. O. Box 1328, Boca Raton, Florida 33432. IBM may use or 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.

© Copyright International Business Machines Corporation 1984

Summary of Changes For Version 5.0

The following changes have been made to this document in addition to editorial updates.

- Control of 4975-01A ASCII Printer operations has been described in the PRINTEXT section of this manual. For that information refer to "Request Special Terminal Function (4975-01A)" on page LR-334.
- A program has been included which enables you to change address(s) for the image and/or control stores of the 4980 Display Station from an application program. A description can be found under "$RAMSEC - Replace Terminal Control Block (4980)" on page LR-594.
- A new operand has been added to the description of the instruction "BSCOPEN - Prepare a BSC line for use" on page LR-41. This operand is for the X.21 Circuit Switched Network.
- Information has been included on how to code a disk immediate read in the READ section of the manual. This information may be found under "Coding Example - Disk Immediate Read" on page LR-381.
- Descriptions of the following new instructions or statements have been added to Chapter 2:
  - "MECB - Create a list of events" on page LR-269. This statement is used to generate an ECB list for the WAITM instruction.
  - A new TERMCTRL statement for the 4980 Display Terminal. This description is found under "4980 Display" on page LR-470.

Summary of Changes For Version 5.0

- A new TERMCTRL statement for the 5219 Printer. This description is found under "5219 Printer" on page LR-473.
- "WAITM - Wait for one or more events in a list" on page LR-523. This instruction allows a program to wait for one or more events in a list.

About This Book

This book contains details and examples of how to code the instructions and statements you can use to write Event Driven Language application programs.

Audience

This book is intended for application programmers who write and maintain programs using the Event Driven Language. You can learn the Event Driven Language by using the *Event Driven Executive Language Programming Guide.*

How This Book Is Organized

This book contains two chapters and six appendixes:

- *Chapter 1. Introduction* describes how instructions and statements are presented in this book. The chapter also describes the syntax rules for the language, defines key terms used throughout the book, and provides information about a number of special features available with the Event Driven Language.
- *Chapter 2. Instruction and Statement Descriptions* contains a detailed description of each EDL instruction and statement and shows the syntax of the instruction or statement, the required operands, and the default values. The instructions and statements are arranged in alphabetical order.

How This Book Is Organized *(continued)*

- *Appendix A. Formatted Screen Subroutines* contains a description of each of the formatted screen subroutines ($IMAGE routines) along with its syntax, required operands, and default values.

- *Appendix B. Programs Communication Through Virtual Terminals* contains a description of the virtual terminal facility that allows application programs to communicate as if they were EDX terminals.

- *Appendix C. Communicating with Programs in Other Partitions (Cross-Partition Services)* contains examples that show how programs can share data and communicate with other programs across partitions.

- *Appendix D. EDX Programs, Subroutines, and Inline Code* lists the syntax, options and default values for the Indexed Access Method, Multiple Terminal Manager, and Formatted Screen subroutines. In addition, the appendix describes a data management program and subroutines, a program for using partitioned data sets, and a copy code routine for identifying device types.

- *Appendix E. Creating, Storing, and Retrieving Program Messages* describes how to build and use formatted program messages in your EDL application programs.

- *Appendix F. Conversion Table* contains a table that shows the hexadecimal, binary, EBCDIC, and ASCII equivalents of decimal values. The table also shows transmission codes for communications devices.

Aids in Using This Book

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

- An Instructions and Statements Chart that groups EDL instructions and statements by the common tasks they perform. The chart also lists the statements used during system generation.

- A Glossary that defines terms and acronyms used in this book and in other EDX library publications.

- An Index of topics covered in this book.

A Guide to the Library

Refer to the *Library Guide and Common Index* for information on the design and structure of the Event Driven Executive Library and for a bibliography of related publications.

Contacting IBM about Problems

You can inform IBM of any inaccuracies or problems you find when using this book by completing and mailing the **Reader's Comment Form** provided in the back of the book.

If you have a problem with the Series/1 Event Driven Executive services, you should fill out an authorized program analysis report (APAR) form as described in the *IBM Series/1 Software Service Guide,* GC34-0099.

Contents

Contents

Contents

Contents

Figures

Chapter 1. Introduction

The Event Driven Language (EDL) is a programming language designed for use on the Series/1 computer. The language enables you to write programs that perform specific tasks. This chapter describes how the various instructions and statements that make up the Event Driven Language are presented in this book. The chapter also includes:

- Definitions of terms commonly used throughout the book
- A list of syntax rules you need to know to code EDL instructions and statements
- A description of how to use parameter naming operands and the two software registers available to your program.

Note: For a detailed description of how to write and structure EDL programs, see the *Event Driven Executive Language Programming Guide*.

The Event Driven Language

The Event Driven Language is composed of **instructions** and **statements**. Instructions allow you to perform specific operations such as adding or subtracting data or printing a message on a terminal. Instructions generate object code that the system can process and execute. Statements enable you to define the parts of a program, define data and system resources, and format compiled output, but not all EDL statements generate object code. The system typically uses the code that is generated by statements to set up storage locations.

Because statements do not execute in the same manner as instructions, you should not place statements between the instructions in your programs. The exception to this rule is the four

The Event Driven Language *(continued)*

statements used to control the formatting of compiler listings: PRINT, SPACE, TITLE, and EJECT. You can code these statements between program instructions because the system ignores them after the compile operation.

The Format of EDL Instructions and Statements

EDL instructions and statements have the general format:

```
label          operation          operands
```

where these terms have the following meanings:

label The symbolic name you assign to an instruction or statement. You can use this name in your program to refer to that specific instruction or statement. In most cases, a label is optional.

operation The name of the instruction or statement you are coding.

operands These constitute the body of the instruction or statement. An operand can represent data that is required to complete an operation, or it can define how an operation is to be performed.

The Event Driven Language has two types of operands: **positional** operands and **keyword** operands. Positional operands must be coded in the position shown in the operands field for the instruction or statement. These operands appear in lower case. Positional operands usually require a specific value, address, or label. Keyword operands can be coded in any order following the positional operands (if any) contained in an instruction or statement. These operands are in the form KEYWORD=. Keyword operands typically enable you to control how the system performs an operation.

Depending on the type of operation you are performing, you may need to code an operand with a specific value or label. For the purposes of this book, such values or labels are generally referred to as **parameters**. Figure 1 shows the syntax of the EDL ADD instruction.

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

Figure 1. ADD Instruction Syntax

In the following example, operand 2 (a value of 5) is added to operand 1 (the contents in A). The system places the result of this operation in SUM, the location specified on the keyword operand RESULT=.

```
                .
                ADD      A,5,RESULT=SUM
                .
A               DATA     F'8'
SUM             DATA     F'0'
                .
```

The parameter for opnd1 in the above operation is A. The parameter specified for opnd2 is 5, and SUM is the parameter coded for the RESULT= operand.

Instruction and Statement Descriptions

This book describes each EDL instruction and statement beginning in Chapter 2. Each description begins with an explanation of what the instruction or statement does. This explanation is followed by a syntax box which shows the operands that make up the instruction or statement. Positional operands are shown in the order you must code them.

Each syntax box also contains a list with the following headings:

Required: You are required to code the operand or operands listed here.

Defaults: The system will supply the data shown if you do not specify the operand or operands listed here.

Indexable: You can use the two software registers, #1 and #2, for the operands listed here. See "Software Register Usage" on page LR-10 for further information on the software registers.

All operands that make up an instruction or statement are defined in a list which follows the syntax box. The operands are listed in the order in which they appear in the syntax box. The operand description details the use of the operand and any restrictions that may apply to its use.

Special Considerations

Certain IBM devices may require you to code an EDL instruction in a special way. Other devices offer additional features which expand the use of an instruction. Special considerations that can affect the way you use an instruction are described after the operand list.

Syntax Examples

Most instructions and statements in this book contain syntax examples. Syntax examples show the various ways you could code an instruction or statement. They generally consist of a single line of code.

Coding Examples

Many instructions and statements in this book also contain one or more coding examples. These examples consist of entire programs or pieces of programs. Coding examples illustrate how an instruction or statement works in relation to other instructions and statements in the language.

The Format of EDL Instructions and Statements *(continued)*

Return or Post Codes

If an instruction issues return or post codes, these are listed after the examples. Return and post codes are issued as follows:

Return codes Issued as a result of executing an EDL instruction to indicate whether the operation was a success or a failure. Return codes are returned in the first word of the task control block of the program or task issuing the instruction, unless otherwise stated. The label of the task control block (TCB) is the taskname (label) you specify on the PROGRAM or TASK statement. You can examine the return code from an instruction by referring to the taskname in your program or by using the TCBGET instruction.

The following example shows several ways you can check the return code:

```
START    PROGRAM   BEGIN
BEGIN    EQU       *
         READTEXT  ...
         IF        (START,EQ,-1),GOTO,MESSAGE
         TCBGET    RC,$TCBCO
         PRINTEXT  'ERROR RETURN CODE IS: '
         PRINTNUM  RC
           .
           .
           .
MESSAGE  PRINTEXT  'OPERATION IS SUCCESSFUL'
           .
           .
           .
RC       DATA      F'0'
```

Post codes Issued by the system to signal the occurrence of an event. Unless otherwise stated, post codes are returned in the first word of the event control block (ECB) that is posted when the event occurs. You must specify the ECB to be posted with an ECB statement.

The Format of EDL Instructions and Statements *(continued)*

Sample EDL Instruction

The following example shows how instructions and statements are presented in this book. A full description of the MESSAGE instruction and its operands appears in Chapter 2.

MESSAGE - Retrieve a program message

The MESSAGE instruction retrieves a program message from a data set or module, and displays or prints the message.

Syntax:

```
label        MESSAGE  msgno,COMP=,SKIP=,LINE=,SPACES=,
                      PARMS=(parm1,...,parm8),MSGID=,
                      XLATE=,PROTECT=,P1=

Required:    msgno,COMP=
Defaults:    MSGID=NO,XLATE=YES,PROTECT=NO
Indexable:   none
```

Operand	**Description**
msgno	(positional operand)
COMP=	(keyword operand)
SKIP=	(keyword operand)
LINE=	(keyword operand)
SPACES=	(keyword operand)
PARMS=	(keyword operand)
MSGID=	(keyword operand)
XLATE=	(keyword operand)
PROTECT=	(keyword operand)
P1=	(parameter-naming operand)

The Format of EDL Instructions and Statements *(continued)*

Syntax Example

Retrieve the first message in the disk data set that the COMP statement points to.

```
MSG1     MESSAGE   1,COMP=MSGSET
           .
           .
           .
         PROGSTOP
MSGSET   COMP      'ERRS',DS1,TYPE=DSK
```

Coding Example

The following example uses the MESSAGE instruction to retrieve a message contained in a disk data set. The program TASK loads a second program CALCPGRM. A WAIT instruction suspends the execution of TASK until CALCPGRM completes. When CALCPGRM finishes, it posts the ECB at label LOADECB. The MESSAGE instruction at label MSG1 retrieves the first message in the disk data set MSGDS1 on volume EDX002.

```
TASK      PROGRAM   START,DS=((MSGDS1,EDX002))
LOADECB   ECB
START     EQU       *
            .
            .
            .
          LOAD      CALCPGRM,EVENT=LOADECB
          WAIT      LOADECB
MSG1      MESSAGE   1,COMP=MSGSET,SKIP=1,PARMS=A,MSGID=YES
            .
            .
            .
          PROGSTOP
A         DATA      'CALCPGRM'
MSGSET    COMP      'STAT',DS1,TYPE=DSK
          ENDPROG
          END
```

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
301-316	Error while reading message from disk.
	.
	.
	.
335	Disk messages not supported (MINMSG support only)

Common Terms

The following list contains some terms commonly used in the *Language Reference*, along with their definitions:

constant A value or address that remains unchanged throughout program execution. The number 5 is an example of an integer constant. An address in a program, such as 009E, is an example of an address constant.

self-defining term A decimal, integer, or character that the computer treats as data and not as an address or pointer to data in storage. Self-defining terms include expressions such as C'A' and X'5B'.

variable An area in storage, referred to by a label, that can contain any value during program execution. In the example below, the label A refers to an area in storage. The area contains the value 10. When the DIVIDE instruction executes, it divides the contents of A by 5. The system places the result of the operation in A. The variable A now contains a value of 2.

```
          DIVIDE  A,5
            .
            .
            .
A         DATA    F'10'
```

immediate data Immediate data refers to the way you can use a self-defining term. If you code a self-defining term, such as 8, for an operand in an instruction, you are using this term as "immediate data." Operand 2 in the following example uses immediate data. The MULTIPLY instruction multiplies the value of B by 8.

```
          MULTIPLY  B,8
```

precision The number of words in storage needed to contain a value in an operation.

Syntax Rules

This section contains syntax rules you should be aware of when coding programs in the Event Driven Language. These rules apply whether you are using the Event Driven Executive Compiler ($EDXASM) or the IBM Series/1 Macro Assembler ($S1ASM).

- An "alphabetic string" can contain one or more alphabetic characters (A - Z) and any of the following special characters: $, #, or @.
- An "alphameric string" can contain one or more alphabetic or numeric characters (0 - 9).

- You must code all instructions, statements, and keyword operands in upper case letters (as shown in the syntax descriptions starting in Chapter 2, "Instruction and Statement Descriptions" on page LR-17).

- When you code a keyword operand, you must also code the equal sign (=) that follows it as shown in the following example.

```
PREC=
```

- Operands must be separated by commas. Operands also must be separated from the operation name by one or more blanks.

- An ellipsis (...) indicates that an operand may be repeated a variable (n) number of times.

- A vertical bar (|) between two operands indicates that you can use one operand or the other, but not both.

- All labels must be alphameric strings of 1 to 8 characters in length. The first character of the label must be a letter or one of the following special characters: $, #, or @.

- Instruction and statement labels must begin in column 1. Operation names can begin in column 2, but must not go beyond column 71.

- To continue a line of code on another line, code any nonblank character in column 72, for example an "X", and begin the next line in column 16. If the continuation line contains a blank between column 16 and column 71, the system ignores any information after that blank. The system concatenates the data on the continuation line to the data on the preceding line.

  The number of continuation lines allowed is limited only by the maximum of 254 characters allowed in the operands field.

  You can code operands through column 71 of the line to be continued, or you can break off the line after a comma following an operand. An example of breaking off the line before column 71 follows:

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7--

label    PRINTEXT   'ANNUAL STATUS AND RECOMMENDATION REPORT',          X
               SPACES=20,SKIP=1
```

- To include a comment following an instruction in your program, separate the comment from the operands field by at least one blank. You can reserve an entire line in the program for comments by coding an asterisk (*) in column 1. The system ignores everything on the line following the asterisk.

Avoid the use of commas within comments for any of the following instructions or statements: DEQT, ECB, ENQT, IOCB, PROGSTOP, or QCB.

- The system interprets any label you assign a value to with the EQU statement as an address unless you code a plus sign (+) in front of the label. The plus sign indicates that the label represents a numeric value.

- 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 subroutine)
  - SETBUSY
  - SUPEXIT
  - SVC

  Note: You can refer to these labels within your program in the instruction operands.

- The maximum number of delimiters allowed in the operands field is 70. Delimiters are ( ) or , or ' .

- To indicate an apostrophe mark within a text message, code double apostrophe marks ('').

- The EDX arithmetic operators are + (plus), - (minus), * (multiply), and / (divide).

  You can use the plus and minus operators to create expressions that refer to specific addresses in your program. The expression B+2, for example, defines an address equal to the address of B plus 2 bytes. The expression C-A defines an address equal to the address of C minus the address of A. You can use the expressions you create with the plus and minus operators in all EDL instructions that allow you to code a label for an operand. You can use an expression instead of a label.

  The multiply and divide operators are valid only when you use them in an arithmetic expression that you equate with a label. You equate arithmetic expressions with labels by using the EQU instruction. The multiply operator multiplies an address by the number of bytes you specify. The expression B*2 multiplies the address of B by 2. The divide operator divides an address by the number of bytes you specify. In the expression C/D, the address of C is divided by the value of D. See the EQU statement for examples that use the multiply and divide operators.

Syntax Rules *(continued)*

Each arithmetic expression can contain only one operator. For example, the expressions A+B, C-1, D*4, and E/2 are all valid. If you require an expression containing more than one operator, you can code it using multiple equate (EQU) statements. The EQU statement equates a label with a value. To compute the address of A+B-2, for example, you could code the following:

```
APB       EQU       A+B               EQUATE APB WITH A+B
APBM2     EQU       APB-2             EQUATE APBM2 WITH APB-2
```

An arithmetic expression normally consists of two terms separated by an operator. You can construct an expression, however, consisting of an operator followed by a symbol. In this case, the system assumes that the first term of the expression is 0. For example, if the value 2 is at location A, then +A is 2, -A is -2, *A is 0, and /A is 0.

- Operands which do not belong with an instruction are normally not flagged as errors when compiled under $EDXASM. The erroneous operand does not generate any code and, therefore, does not affect the execution of the instruction.

Software Register Usage

Each task in your program has access to two software registers. You can use these registers to hold data during an operation or as a means of computing addresses. You can also use the registers as counters. The registers are named #1 and #2. With operands that are listed as "indexable," you can treat the registers in the same manner as any other variable. For example, you can code instructions in your program to set, modify, or test these registers.

In the example below, the MOVE instruction moves the value 0 into #1. The 0 value replaces any existing data in #1, thereby setting the software register to 0.

```
MOVE    #1,0            SET #1 TO ZERO
```

The MOVE instruction in the next example moves the contents of variable A into #2.

```
MOVE    #2,A            SET #2 TO THE CONTENTS OF A
```

An example of a register used as the second operand in an instruction is:

```
ADD     A,#1
```

Here, the ADD instruction adds the contents of #1 to the variable A, and places the result in A.

You may also want to place the address of a variable into a software register. You can accomplish this by using the MOVEA instruction. For example,

```
MOVEA    #2,BUFFER1
```

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

Indexing with the Software Registers

You can use #1 and #2 to modify addresses in your program while the program is executing. The process is called "indexing" and #1 and #2 are referred to as "index registers." In the following example,

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

the MOVE instruction moves the contents specified by (B,#1) into variable A. The system treats the second operand of the MOVE instruction as an address because this operand is in the form,

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

where *parameter* is either a label or an integer and *r* is either a 1 or a 2. If #1 in the preceding example contains a 5, then the data the system moves into variable A is located at the address of B plus 5 bytes. This sum is called the "indexed address." Note that only one of the variables in an operand with the (parameter,#r) format, either the parameter or the index register, can represent an address. The other variable must be an integer or a label preceded by a plus sign (+) that is equated to an integer. (Use the EQU statement to equate a label with an integer.)

The following example shows how you could use an index register to find the location of data in a buffer. The example uses a DO loop to find the value -350 in a buffer containing 1000 entries.

```
          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
          .
          .
          PROGSTOP
BUF       BUFFER   1000,WORDS
```

The first MOVE instruction sets the index register, #1, to 0. A DO instruction is coded to perform the operations within the loop 1,000 times. The IF instruction checks to see if the first word in the buffer BUF is equal to -350. If the first word is not equal to -350, the ADD instructions adds the value 2 to #1. When the loop repeats, (BUF,#1) points to the address of

Software Register Usage *(continued)*

BUF plus two bytes (one word). With each succeeding loop, the program increments #1, and points to the next word in the buffer. BUF has a length of 1,000 words (2,000 bytes).

If the program finds the value -350 in the buffer, it executes the MOVE instruction at label FOUND. The MOVE instruction saves the displacement from the start of the buffer, which is contained in #1, at the location DISP.

Register Considerations

Because each task in a program has its own software registers, the values in #1 and #2 can vary from task to task. The system will use whatever values are in the software registers of the task that is executing.

If several different tasks call a subroutine, the subroutine uses the software registers belonging to the calling task. Overlay programs, however, are independent programs with their own tasks. They have their own registers and do not use the invoking task's registers.

Using The Parameter Naming Operands (Px=)

Often, when you create a program, you do not know the exact data the program will use when it executes. Normally, you can code a label with a DATA, DC or TEXT statement. In the MOVE instruction, for example, you may not know the byte count until a previous instruction executes. When the instruction executes, it uses whatever data is stored at the location defined by the label. Sometimes, however, a label cannot be coded for instruction parameters.

In the following example, the number of bytes to move is dependent on the value of the variable called NUMBER. The count parameter of the MOVE instruction does not allow use of a label. So, multiple MOVE instructions are needed for every count parameter option. In the following example, only two values for NUMBER exist. A separate MOVE instruction is needed for each value. Note that this technique requires a great deal of storage.

```
              .
              .
          IF (NUMBER,EQ,6)
            MOVE A,B,(6,bytes)
          ELSE
            IF (NUMBER,EQ,10)
              MOVE A,B,(10,bytes)
            ENDIF
          ENDIF
              .
              .
A         TEXT    LENGTH=10
B         TEXT    LENGTH=10
NUMBER    DATA      F'0'
```

If the value of NUMBER is a 6, then 6 bytes are moved from location B to A. If the value of NUMBER is 10, 10 bytes are moved from location B to A.

The parameter naming operand (Px=) enables you to supply data to an instruction in your program without having to define it with a DATA, DC or TEXT statement.

The Px= operands correspond to other operands in the instruction syntax. P1= represents the first operand in an instruction, P2= represents the second operand, P3= represents the third operand, and so on. The number of parameter naming operands allowed within each instruction varies.

Figure 2 shows the syntax for the MOVE instruction. The MOVE instruction has three parameter naming operands. P1= refers to *opnd1*, P2= refers to *opnd2*, and P3= refers to *count*.

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

Figure 2. MOVE Instruction Syntax

To use a Px= operand, you must first code it with a label. The label refers to a storage location within the instruction. The system refers to the label you assign to the Px= operand when your program executes. The system treats the label as the parameter of the operand to which the Px= operand refers. Once you assign a label to the Px= operand, you can use that label in other instructions in your program.

In the following example, a parameter naming operand (P3=) is used on the MOVE instruction to provide the number of bytes to be moved.

```
                  .
                  .
         MOVE     A,B,(0,bytes),P3=NUMBER
                  .
                  .
A        TEXT     LENGTH=10
B        TEXT     LENGTH=10
                  .
                  .
```

This single line of code can replace the previous example. The system generates the label and data area NUMBER when it assembles the MOVE instruction. The count parameter of the MOVE instruction updates automatically when the variable called NUMBER contains the value 6 or 10. This method of coding does not require an IF instruction because the NUMBER variable is in the MOVE instruction. The system generates the variable called NUMBER from the Px= operand code. Storage is significantly reduced because it uses only one MOVE instruction.

In the following program, the GETVALUE instruction asks you for the number of bytes to move from B to A. Since the TEXT statement is only 10 bytes, the program checks for errors in

Using The Parameter Naming Operands (Px=) *(continued)*

data by making sure INPUT is between 1 and 10 bytes. When the GETVALUE instruction receives the value for INPUT, the system automatically updates the MOVE instruction's byte count field. At that point the data and characters moved from location B to A are printed on the terminal.

```
TEST      PROGRAM    START
START     EQU        *
RETRY     GETVALUE   INPUT,MESSAGE
          IF         (INPUT,LT,0),or,(INPUT,GT,10),GOTO,RETRY
          MOVE       A,B,(0,bytes),P3=INPUT
          PRINTEXT   A
          PRINTEXT   SKIP=1
          PROGSTOP
A         TEXT       '          ',LENGTH=10
B         TEXT       'ABCDEFGHIJ',LENGTH=10
MESSAGE   TEXT       'ENTER BYTE COUNT'
          ENDPROG
          END
```

Using The Parameter Naming Operands (Px=) *(continued)*

Rules to Remember

You should remember the following rules when coding parameter naming operands in your program.

Coding labels on Px= operands

When the compiler sees a Px= operand, it generates the label that you specify. The compiler flags an error if you attempt to define that label again in your program.

Referring to Px= operand labels

You can refer to the label you code on the Px= operand more than once in your program. However, once you have defined a label with a Px= operand, you cannot use the same label on another Px= operand in the program.

Coding the operand that Px= replaces

When you code a Px= operand, you must still code a value or label for the operand that Px= replaces. The system does not process the Px= operand if the label you specified for it contains a 0 when the instruction executes. (The system defines the value of the label on the Px= operand to be 0 at compilation time.) The example that follows shows a case in which the system does not process the P2= operand until the instruction at GETDATA executes and supplies label B with a value other than 0.

```
CHECK     PROGRAM   START
START     EQU       *
ADDVAL    ADD       A,0,P2=B
          IF        (A,GT,10),GOTO,END
GETDATA   GETVALUE  B,'ENTER NUMBER FROM 1 TO 10 ',SKIP=1
          GOTO      ADDVAL
END       PRINTNUM  A,SKIP=1
          PROGSTOP
A         DATA      F'1'
          ENDPROG
          END
```

On the first pass through the program, the label B contains a 0. The system adds the value coded for operand 2 (0) to the value in A. After the GETVALUE instruction executes, B contains whatever value was entered at the terminal. The GOTO instruction passes control to the ADD instruction at the label ADDVAL. When the ADD instruction executes the second time, the system adds the value in B to the value in A. The system replaces the 0 value coded for operand 2 with the value entered in B.

Using The Parameter Naming Operands (Px=) *(continued)*

Matching operand and Px= operand data types

The type of data that the Px= operand supplies in an instruction must match the type of data that is being replaced. For example, if you specify the label of an *address* for operand 2, P2= must also supply an *address*. If you specify a *constant* for operand 2, P2= must supply a *constant*.

In the example that follows, the ADD instruction contains a P2= operand. The P2= operand refers to operand 2 which is coded with the constant 5. Because the parameter coded for operand 2 is a constant, the P2= operand must replace this parameter with another constant to get the desired results. In this case, the MOVE instruction moves the value 2 into A. The system adds 2 to C and stores a result of 2 in SUM.

```
           .
        MOVE    A,2
        ADD     C,5,RESULT=SUM,P2=A
           .
           .
C       DATA    F'0'
SUM     DATA    F'0'
           .
           .
```

In the next example, operand 2 of the ADD instruction is coded with the label D. The label refers to the address of a data area. Because the parameter coded for operand 2 (D) is an address, the P2= operand must replace this parameter with another address to get the desired results. In this case, a MOVEA instruction moves the address of B into A. The system adds the contents of B to the contents of C and places the result in SUM.

```
           .
        MOVEA   A,B
        ADD     C,D,RESULT=SUM,P2=A
           .
           .
B       DATA    F'2'
C       DATA    F'0'
D       DATA    F'5'
SUM     DATA    F'0'
```

Chapter 2. Instruction and Statement Descriptions

This chapter presents the Event Driven Language (EDL) instructions and statements in alphabetical order. A description of the use of each instruction and statement is provided, followed by its syntax, required operands, and the default values the system uses when you do not specify certain operands. Each operand is listed and described. Examples and other information, such as return codes and post codes, also are provided. See "The Format of EDL Instructions and Statements" on page LR-2 for more details on how this book presents instructions and statements.

Note: The *Installation and System Generation Guide* contains the statements you use to define and generate your system. These statements are listed in the "Instructions and Statements Chart."

Instructions and Statements Chart

The chart on the following pages groups EDL instructions and statements by the common tasks they perform. The chart also lists the statements you use to define and generate a system.

Instructions and Statements Chart *(continued)*

Add Device Support	Define Data
DCB EXOPEN EXIO IDCB	ALIGN EQU BUFFER STATUS DATA/DC TEXT
Call Programs and Subroutines	Define I/O
CALL RETURN CALLFORT USER SUBROUT	BSCIOCB IODEF CAIOCB PROGRAM IOCB SBIO
Code Graphics Applications	End a Program
CONCAT SCREEN GIN XYPLOT PLOTGIN YTPLOT	END ENDPROG PROGSTOP
Control Program Logic	Format and Identify Compiler Listings
DO FINDNOT ELSE GOTO ENDIF IF ENDDO QUESTION FIND	$ID SPACE EJECT TITLE PRINT
Control Tasks	Initiate and Terminate Telecommunications
ATTACH LOAD ATTNLIST PROGRAM DETACH PROGSTOP END QCB ENDATTN RESET ENDPROG TASK ENDTASK WHERES	BSCCLOSE NETHOST BSCOPEN NETINIT CACLOSE NETTERM CAOPEN TP CLOSE CASTART TP OPENIN CASTOP TP OPENOUT NETCTL
Control the Terminal	Manipulate Data
ATTNLIST IOCB ENDATTN RDCURSOR ERASE TERMCTRL	ADD FSUB ADDV HASHVAL AND IOR
Convert Data	CONCAT MOVE DIVIDE MOVEA
CONVTB FPCONV CONVTD GETEDIT FORMAT PUTEDIT	EOR MULTIPLY FADD SETBIT FDIVD SHIFTL FMULT SHIFTR FPCONV SQRT SUBTRACT

Obtain Date and Time	Respond to Errors
GETTIME PRINDATE PRINTIME	CATRACE SBIO FREESTG SWAP GETEDIT TCBGET GETSTG TCBPUT LOAD WRITE READ
Obtain and Release Resources	**Retrieve User-Written Messages**
DEQ DEQT ENQ ENQT FREESTG GETSTG STORBLK SWAP	COMP QUESTION GETVALUE READTEXT MESSAGE **Refer to External Modules** COPY EXTRN CSECT WXTRN ENTRY
Perform Communication I/O	**Send or Receive Terminal Data**
CAREAD TP (READ) CAWRITE TP (RELEASE) CAPRINT TP (SET) NETGET TP (SUBMIT) NETPUT TP (WRITE) TP (FETCH)	GETEDIT PRINTEXT GETVALUE PRINTIME MESSAGE PUTEDIT PRINDATE QUESTION PRINTNUM READTEXT
Perform Disk, Diskette, and Tape I/O	**Set Timers**
CONTROL POINT DSCB READ NOTE WRITE	INTIME STIMER
Process Interrupts	**Synchronize Tasks**
ATTNLIST IODEF SPECPIRT	ECB STIMER INTIME WAIT POST
Queue Processing	**System Generation**
DEFINEQ FIRSTQ LASTQ NEXTQ	ADAPTER SNALU BSCLINE SNAPU DISK SYSTEM EXIODEV TAPE HOSTCOMM TERMINAL SENSORIO TIMER

$ID - Identify system release level

The $ID statement enables you to record within an application program the EDX system release level that you use to compile the program. If you dump the program at a later date to diagnose a problem, the $ID statement eliminates the need to refer back to the original source listing to find out the system release level in use when the program was compiled.

The system release level coded with $ID appears as the last word in the dumped program.

Code the $ID statement between the ENDPROG and and END statements of your program. This is an exception to the rule that ENDPROG and END must be the last two statements of your program.

The $ID statement generates a 1-word constant in the form of 'VMLP'. Each parameter is packed into four bits and is specified in hexadecimal notation.

The $ID statement is already coded on all EDX supplied software.

Syntax:

```
label          $ID        V=,M=,L=,P=

 Required:     None
 Defaults:     V=,M=, and P= default to the current release level
               of the EDX program product
```

Operand	*Description*
V=	The EDX system release level; it ranges from 0-9, A-F (hexadecimal).
M=	The EDX modification or revision level; it ranges from 0-9, A-F (hexadecimal).
L=	The unique identifier you assign to programs not prepared by IBM; it ranges from 1-9, A-F (hexadecimal). The value 0 is reserved for IBM use.
P=	The program temporary fix (PTF) release level; it ranges from 0-9, A-F (hexadecimal).

$ID - Identify system release level *(continued)*

Syntax Examples

1) In the following example, only operand L, which is designated for your use, is coded. Operands V, M, and P are allowed to default to the current release level of the EDX program product.

```
                 .
                 .
                 .
         ENDPROG
IDNOTE   $ID          L=2
         END
```

2) The $ID statement in the example below will cause the identifier, '3121', to be printed out as the last word in the program when it is dumped. The identifier shows that the program was compiled under EDX system release level 3, modification level 1, and program temporary fix 1. The 2 on the L= operand is for the programmer's use.

```
                 .
                 .
                 .
         ENDPROG
IDNOTE   $ID         V=3,M=1,L=2,P=1
         END
```

ADD - Add integer values

The ADD instruction adds an integer value in operand 2 to an integer value in operand 1. The values can be positive or negative. To add floating-point values, use the FADD instruction.

See the DATA/DC statement for a description of the various ways you can represent integer data.

EDX does not indicate an overflow condition for this instruction.

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
```

Operand	***Description***
opnd1	The label of the data area to which opnd2 is added. Opnd1 cannot be a self-defining term. The system stores the result of the ADD operation in opnd1 unless you code the RESULT operand.
opnd2	The value added to opnd1. You can specify a self-defining term or the label of a data area. The value of opnd2 does not change during the operation.
count	The number of consecutive values in opnd1 upon which the system performs the operation. The maximum value allowed is 32767.
RESULT=	The label of a data area or vector in which the result is placed. The data area you specify for opnd1 is not modified if you specify RESULT. This operand is optional.
PREC=xyz	Specify the precision of the operation in the form xyz, where x is the precision for opnd1, y is the precision for opnd2, and z is the precision of the result ("Mixed-precision Operations" on page LR-23 shows the precision combinations allowed for the ADD instruction). You can specify single-precision (S) or double-precision (D) for each operand. Single precision is a word in length; double precision is two words in length. The default for opnd1, opnd2, and the result is single precision. If you code a single letter for PREC, the letter applies to opnd1 and the result. Opnd2 defaults to single precision. If, for example, you code PREC=D, opnd1 and the result are double precision and opnd2 defaults to single precision.

ADD - Add integer values *(continued)*

If you code two letters for PREC, the first letter applies to opnd1 and the result, and the second letter applies to opnd2. With PREC=DD, for example, opnd1 and the result are double precision and opnd2 is double precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Mixed-precision Operations

The following table shows the precision combinations allowed with the ADD instruction:

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

Opnd2 is one or two words long depending on the precision you specify on the PREC= keyword. The length of opnd1 is equal to the operand's precision multiplied by the value of the count operand.

ADD - Add integer values *(continued)*

Coding Example

The following example moves the value 0 to index register #1. Next, the value 5 is added to #1. Index register #1 now contains the value 5. The contents of variable A are then added to each of three words starting at label V1. The results are placed in three words starting at label V2. The contents of V1 and A remain unchanged because the keyword RESULT is specified. The third ADD instruction adds 15 to the double-precision value at label E.

```
                 .
                 .
                 .
         MOVE   #1,0                MOVE 0 TO #1
         ADD    #1,5                INCREASE #1 BY 5
         ADD    V1,A,3,RESULT=V2    ADD THE VALUE IN A TO EACH OF 3 WORDS
*                                   STARTING AT V1 AND PLACE THE RESULT
*                                   IN 3 WORDS STARTING AT V2.
*
         ADD    E,15,PREC=D         ADD 15 TO DOUBLE-PRECISION VALUE E.
*
A        DATA   F'10'
V1       DATA   F'1'
         DATA   F'2'
         DATA   F'3'
V2       DATA   F'0'
         DATA   F'0'
         DATA   F'0'
E        DATA   D'100000'
                 .
                 .
                 .
```

The results from the above coding example follow:

	Before		**After**
#1	F'0'	#1	F'5'
A	F'10'	A	F'10'
V1	F'1'	V1	F'1'
	F'2'		F'2'
	F'3'		F'3'
V2	F'0'	V2	F'11'
	F'0'		F'12'
	F'0'		F'13'
E	D'100000'	E	D'100015'

ADDV - Add two groups of numbers (vectors)

The add vector instruction (ADDV) adds two groups of numbers or "vectors". The number of times the operation occurs depends on the count you specify. The instruction adds each consecutive value in operand 2 to the corresponding value 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:    count=1,RESULT=opnd1,PREC=S
Indexable:   opnd1,opnd2,RESULT
```

Operand	***Description***
opnd1	The label of the data area that is modified by opnd2. Opnd1 cannot be a self-defining term. Do not code the software registers, #1 or #2, for this operand. You can use the software registers, however, to create an indexed address for opnd1.
opnd2	The value by which opnd1 is modified. You can specify a self-defining term or the label of a data area.
count	The number of consecutive values in both opnd1 and opnd2 upon which the system performs the operation. The maximum value allowed is 32767.
RESULT=	The label of a data area or vector in which the result is placed. The data area you specify for opnd1 is not modified if you specify RESULT. This operand is optional.
PREC=xyz	Specify the precision of the operation in the form xyz, where x is the precision for opnd1, y is the precision for opnd2, and z is the precision of the result. ("Mixed-precision Operations" on page LR-26 shows the precision combinations allowed for the ADDV instruction.) You can specify single-precision (S) or double-precision (D) for each operand. Single precision is a word in length; double precision is two words in length. The default for opnd1, opnd2, and the result is single precision. If you code a single letter for PREC, the letter applies to opnd1 and the result. Opnd2 defaults to single precision. If, for example, you code PREC=D, opnd1 and the result are double precision and opnd2 defaults to single precision.

ADDV - Add two groups of numbers (vectors) *(continued)*

If you code two letters for PREC, the first letter applies to opnd1 and the result, and the second letter applies to opnd2. With PREC=DD, for example, opnd1 and the result are double precision and opnd2 is double precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Mixed-precision Operations

The following table lists the precisions allowed with the ADDV instruction:

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

Syntax Example

The ADDV instruction in the following example adds each consecutive value in V1 to the corresponding value in V2. After the instruction executes, V1 contains 32F'3'.

```
          ADDV    V1,V2,32        THE COUNT IS 32
           .
           .
           .
V1        DATA    32F'1'
V2        DATA    32F'2'
```

ADDV - Add two groups of numbers (vectors) *(continued)*

Coding Example

The following example moves the value 10 to X1 and the value 20 to X2. The first ADDV instruction adds the value in C1 to X1 and the value in C2 to X2. Because the keyword RESULT is specified, the values in C1, C2, X1, and X2 remain unchanged. The system places the results in D1 and D2. The second ADDV instruction adds the values of the five words, starting at B1, to the values of the five words starting at A1. The ADDV operation occurs in the following sequence: The value in B1 is added to the value in A1, the value in B2 is added to the value in A2, and so on through B5 and A5.

Results of the example follow on the next page.

```
          .
          .
          .
        MOVE    X1,10                 MOVE 10 TO X1
        MOVE    X2,20                 MOVE 20 TO X2
*
        ADDV    X1,C1,2,RESULT=D1     ADD VALUE OF C1 TO X1 AND
*                                     THEN C2 TO X2

*                                     PLACE RESULTS IN
*                                     LOCATIONS D1 and D2
*
        ADDV    A1,B1,5               ADD THE VALUE OF THE 5 WORDS
*                                     STARTING AT B1 TO THE 5 WORDS
*                                     STARTING AT A1
X1      DATA    F'0'
X2      DATA    F'0'
*
A1      DATA    F'1'
A2      DATA    F'2'
A3      DATA    F'3'
A4      DATA    F'4'
A5      DATA    F'5'
*
B1      DATA    F'10'
B2      DATA    F'20'
B3      DATA    F'30'
B4      DATA    F'40'
B5      DATA    F'50'
*
C1      DATA    F'5'
C2      DATA    F'10'
*
D1      DATA    F'0'
D2      DATA    F'0'
```

ADDV - Add two groups of numbers (vectors) *(continued)*

Results of the previous coding example follow:

	Before		After
X1	F'00'	X1	F'10'
X2	F'00'	X2	F'20'
A1	F'1'	A1	F'11'
A2	F'2'	A2	F'22'
A3	F'3'	A3	F'33'
A4	F'4'	A4	F'44'
A5	F'5'	A5	F'55'
B1	F'10'	B1	F'10'
B2	F'20'	B2	F'20'
B3	F'30'	B3	F'30'
B4	F'40'	B4	F'40'
B5	F'50'	B5	F'50'
C1	F'5'	C1	F'5'
C2	F'10'	C2	F'10'
D1	F'0'	D1	F'15'
D2	F'0'	D2	F'30'

ALIGN - Align instruction or data to a specified boundary

The ALIGN statement ensures that the next instruction or data item in a source statement list begins on a specified boundary: an odd byte, a word, or a doubleword. The ALIGN statement is non-executable and should only be used to align data within data areas.

When coding the ALIGN instruction, you can include a comment which will appear with the instruction on your compiler listing. If you include a comment, you must also code the **type** operand. The comment must be separated from the operand field by at least one blank and it may not contain commas.

Syntax:

```
blank         ALIGN    type    comment

Required:     none
Default:      WORD
Indexable:    none
```

Operand	***Description***
type	WORD (the default) or blank aligns data on a fullword boundary.
	BYTE aligns data on an odd-byte boundary.
	DWORD aligns data on a doubleword boundary.

Note: If the data field is already aligned at the boundary requested, no action results. WORD and BYTE align the data a maximum of 1 byte. DWORD aligns the data a maximum of 3 bytes.

Coding Example

The ALIGN statement in the following example aligns the data area labeled BUFF on a word boundary (even address).

```
Loc

0200     PROGNME   DC      C'EDX UTILITY'
020B               ALIGN   ALIGN TO WORD BOUNDARY
020C     BUFF      DC      CL'64'
```

AND - Compare the binary values of two data strings

The AND instruction compares the binary value of operand 2 with the binary value of operand 1. The instruction compares each bit position in operand 2 with the corresponding bit position in operand 1 and yields a result, bit by bit, of 1 or 0. If both of the bits compared are 1, the result is 1. If either or both of of the bits compared are 0, the result is 0.

Syntax:

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

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

Operand	***Description***
opnd1	The label of the data area to which opnd2 is compared. Opnd1 cannot be a self-defining term. The system places the result of the operation into opnd1 unless you code the RESULT operand.
	The length of opnd1 is equal to the operand's precision multiplied by the value of the count operand.
opnd2	The value compared to opnd1. You can specify a self-defining term or the label of a data area.
count	The number of consecutive values 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. Select one precision which the system uses for opnd1, opnd2, and the resulting bit string. When specifying a precision, code the count operand in the form,

```
(n,precision)
```

where "n" is the count and "precision" is one of the following:

BYTE -- byte precision
WORD -- word precision (default)
DWORD -- doubleword precision

The precision you specify for the count operand is the portion of opnd2 that is used in the operation. If the count is (3,BYTE), the system compares the first byte of data in opnd2 with the first three bytes of data in opnd1.

AND - Compare the binary values of two data strings *(continued)*

RESULT= The label of a data area or vector in which the result is to be placed. When you specify this operand, the value of opnd1 does not change during the operation.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) In the following example, the AND instruction turns off the rightmost four bits in DATA1 without affecting the other data field bits. After the instruction executes, DATA1 contains X'E0' (binary 1110 0000).

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

2) The AND instruction in this example compares opnd2 with the first three bytes of data in opnd1. The system places the result in RESULTX.

```
        AND    OPER1,OPER2,(3,BYTE),RESULT=RESULTX
         .
         .
         .
OPER1   DC     X'00'         binary 0000 0000
        DC     X'A5'         binary 1010 0101
        DC     X'01'         binary 0000 0001
OPER2   DC     X'FF'         binary 1111 1111
RESULTX DC     2F'0'         binary 0000 0000 0000 0000
```

After the AND operation, RESULTX contains X'00A5 0100' (binary 0000 0000 1010 0101 0000 0001).

3) In the following example, the AND instruction compares the first byte of data in TEST to the first three bytes of data in INPUT. The system stores the result in OUTPUT.

```
        AND    INPUT,TEST,(3,BYTE),RESULT=OUTPUT
         .
         .
         .
INPUT   DC     C'1.2'      binary 1111 0001 0100 1011 1111 0010
TEST    DC     C'0.0'      binary 1111 0000 1111 0000 1111 0000
OUTPUT  DC     3C'0'       binary 1111 0000 1111 0000 1111 0000
```

After the AND operation, the contents of OUTPUT are C'0 0' (binary 1111 0000 0100 0000 1111 0000).

ATTACH - Start a task

The ATTACH instruction starts the execution of or "attaches" another task. If the task you specify has already been attached, no operation occurs. You deactivate tasks with the DETACH instruction.

The task to be attached is usually in the same partition as the ATTACH instruction. However, you can attach a task in another partition by using the cross-partition capability of ATTACH.

Note that the program load point of the attaching task is placed in the $TCBPLP field of the task being attached. The system, however, will not reference the $TCBPLP of the attached task if the attaching task is in another partition. To avoid this problem, put the load point of the task to be attached in the $TCBPLP field of the attaching task before the ATTACH instruction is executed. Be sure to restore it after the ATTACH instruction is completed.

See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example of attaching a task in another partition. Refer to the *Event Driven Executive Language Programming Guide* for more information on cross-partition services.

The system records the address space in which a task is executing in the $TCBADS field of the task's task control block (TCB). When your program attaches a task, the system moves the address space in the program's TCB into the $TCBADS field of the attached task's TCB.

When the ATTACH instruction executes, the system stores the address of the terminal from which the main task was loaded in the $TCBCCB field of the attached task. In this way, the same terminal is active for both tasks.

If your program is to be link edited, place all TASKS to attach via the ATTACH instruction in the same module. The assembler will chain all the TASKS within the module it assembles. Your application program will have to chain the tasks together if they are not within the same module. Modify the correct field in the TCB to chain tasks accross modules.

Syntax:

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

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

ATTACH - Start a task *(continued)*

Operand	*Description*
taskname	Label of the task to be attached. You must define this task with a TASK statement.
priority	The priority you assign to the task. This priority replaces the one you assigned on the TASK statement. It remains in effect unless it is overridden by a subsequent ATTACH instruction. See the TASK statement for a description of the valid priorities you can assign a task.
CODE=	A code word to be inserted in the first word of the task control block of the task being attached. This code word could help your program determine at what point the task is being attached. The attached task could examine the code word by referring to the taskname operand. The code word should be examined immediately upon entry into the attached task because execution of certain instructions (for example, I/O instructions) cause this word to be overlaid.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

In the following example, the ATTACH instruction attaches a task that reads a record from a data set. The program begins by attaching TASK1. TASK1 is the label of a TASK statement. TASK1 prints the message at label P1 and reads a record from MYFILE into the buffer BUF. The MOVE instruction moves the first 8 bytes of BUF into the text buffer labeled REC. When TASK1 ends, it posts the event specified on the EVENT= operand of the TASK statement. The main program receives control and the WAIT instruction at label W1 checks to see if TASK1 has ended. The PRINTEXT instruction at label P2 prints the message 'PROGRAM COMPLETE', and the program ends.

```
SAMPLE      PROGRAM     START,DS=((MYFILE,EDX40))
START       EQU         *
            ATTACH      TASK1
W1          WAIT        EVENT
P2          PRINTEXT    'PROGRAM COMPLETE',SKIP=2
            PROGSTOP
BUF         BUFFER      256,BYTES
REC         TEXT        LENGTH=8
*************************************************
TASK1       TASK        NEXT,EVENT=EVENT
NEXT        ENQT        $SYSPRTR
P1          PRINTEXT    '@TASK1 ATTACHED'
            READ        DS1,BUF,1
            MOVE        REC,BUF,(8,BYTES)
            DEQT        $SYSPRTR
            ENDTASK
*************************************************
            ENDPROG
            END
```

ATTNLIST - Enter attention-interrupt-handling routine

The ATTNLIST statement provides entry to one or more attention-interrupt-handling routines.

With the ATTNLIST statement, you can produce a list of command names and associated routine entry points. When you press the attention key on a terminal, your program waits for you to enter a 1–8 character command. If the command you enter matches one that is specified in the list, the associated routine receives control. No action occurs if the command you enter is not contained in the list or if the system cannot find the entry point of the routine.

The character $ is reserved for system use and should not be used as the first character of a command name unless you are assigning PF keys. All other character combinations are allowed. Your attention routines must end with an ENDATTN instruction.

Your program and the ATTNLIST routine execute asynchronously. When the ATTNLIST routine finishes, control passes to the instruction that was executing when you pressed the attention key. Figure 3 on page LR-37 shows the operation of the ATTNLIST instruction.

The attention list for programs you compile with $EDXASM can be up to 254 characters long and can contain a total of 24 ATTNLIST entries. A program compiled under $EDXASM can contain one LOCAL ATTNLIST statement and one GLOBAL ATTNLIST statement. (See the SCOPE= operand for an explanation of LOCAL and GLOBAL ATTNLIST.) The Series/1 macro assembler and the host assembler allow multiple attention lists with a maximum of 125 characters in each list.

ATTNLIST routines should execute quickly. Because the routines execute on hardware level 1, lengthy routines can slow the execution of other application programs or system tasks.

Notes:

1. You should not use the following instructions in an ATTNLIST routine: DETACH, ENDTASK, PROGSTOP, LOAD, STIMER, WAIT, TP, READ, WRITE, ENQT, and DEQT.

2. ATTNLIST routines cannot gain access to an enqueued terminal until the program that has exclusive access releases the terminal by issuing a DEQT or PROGSTOP instruction.

3. Do not use $DEBUG command names as command names in your attention list routine. Refer to the *Operator Commands and Utilities Reference* for a list of the $DEBUG command names.

Syntax:

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

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

ATTNLIST - Enter attention-interrupt-handling routine *(continued)*

Operand	***Description***
cc1	A command name consisting of 1–8 alphameric characters. Do not use the character $ as the first character of the command name unless you are assigning PF keys. For a description of using and assigning the 4979, 4978, 4980, and 3101 terminal program function (PF) keys to invoke ATTNLIST routines, refer to the *Operation Guide*.
loc1	Name of the routine to be invoked.
SCOPE=	GLOBAL, allows the ATTNLIST command routines to be invoked from any terminal assigned to the same storage partition. LOCAL, limits the invoking of ATTNLIST commands to the specific terminal (assigned to the same partition) from which the program containing the commands was loaded. A program may have one LOCAL ATTNLIST and one GLOBAL ATTNLIST.

Syntax Example

The ATTNLIST statement that follows allows you to invoke the PCODE1 routine by pressing the attention key and entering PC1. To invoke the PCODE2 routine, you would press the attention key and enter PC2.

```
          ATTNLIST    (PC1,PCODE1,PC2,PCODE2)
            .
            .
PCODE1    MOVE        CODE,1
          ENDATTN
            .
            .
PCODE2    POST        EVENT,2
          ENDATTN
```

ATTNLIST - Enter attention-interrupt-handling routine *(continued)*

Coding Examples

1)The following example uses the ATTNLIST statement to control the printing of repetitive test patterns. Once the test pattern begins printing, it can only be stopped by pressing the attention key and entering the command "CA".

The program begins printing a test pattern consisting of 10 numbers. You can expand the test pattern to include 24 special characters by pressing the PF1 key.

If you press the PF2 key, the test pattern includes the alphabet, the 10 numbers (0-9), and the 24 special characters.

```
TESTLOOP     PROGRAM    START
             ATTNLIST   (CA,CANCEL,$PF1,PF1,$PF2,PF2)
CANCEL       EQU        *
             MOVE       SWITCH,99
             ENDATTN
PF1          EQU        *
             MOVE       SWITCH,1
             ENDATTN
PF2          EQU        *
             MOVE       SWITCH,2
             ENDATTN
START        EQU        *
             ENQT
             DO         WHILE,(SWITCH,NE,99)
                PRINTEXT    '@1234567890'
                IF          (SWITCH,GE,1)
                PRINTEXT    '|#$%¢&*()_-+=!¬":;?/>.<,'
                ENDIF
                IF          (SWITCH,EQ,2)
                PRINTEXT    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
                ENDIF
             ENDDO
             DEQT
             PROGSTOP
SWITCH       DATA          F'0'
             ENDPROG
             END
```

ATTNLIST - Enter attention-interrupt-handling routine *(continued)*

2)The following example also illustrates coding of the ATTNLIST statement. It, however, uses PF keys to invoke ATTNLIST instead of entering a command.

```
ATTEST          PROGRAM         ATLIST
                ATTNLIST        ($PF1,PCODE1,$PF3,PCODE3)
PCODE1          PRINTEXT        'PF1 KEY WAS PRESSED@'
                MOVE            VAR,1
                ENDATTN
PCODE3          PRINTEXT        'PF1 KEY WAS PRESSED@'
                MOVE            VAR,3
                ENDATTN
ATLIST          EQU             *
                DO              (WHILE,(VAR,NE,1)
                MOVE            #1,#2
                ENDDO
                PROGSTOP
VAR             DATA            X'0000'
                ENDPROG
```

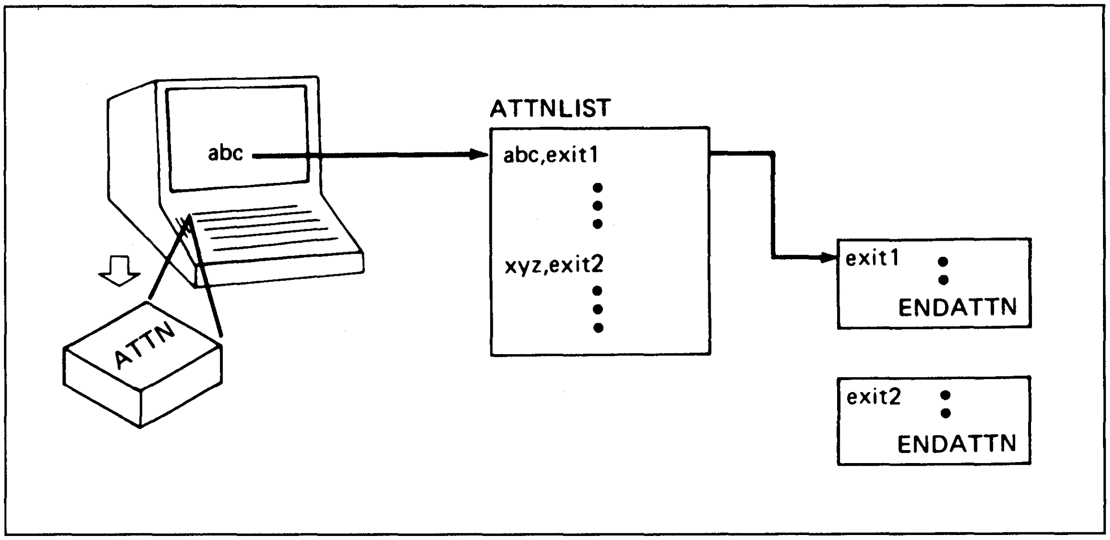

Figure 3. Function of ATTNLIST

BSCCLOSE

BSCCLOSE - Free a BSC line for use by other tasks

The BSCCLOSE instruction frees a binary synchronous line for use by other tasks. If the line is a switched line (TYPE=SM or SA), this instruction disconnects it.

Syntax:

```
label         BSCCLOSE  bsciocb,ERROR=,P1=,P2=

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

Operand	*Description*
bsciocb	The label or indexed location of the BSCIOCB statement associated with the close operation.
ERROR=	The label of the instruction to be executed if an error occurs while closing the line. If you do not code this operand, control passes to the next sequential instruction. In either case, the return code reflects the results of the operation.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All BSC instruction return codes are listed with the BSCWRITE instruction under "Return Codes" on page LR-54.

BSCIOCB - Specify BSC line address and buffers

The BSCIOCB statement specifies the line address and buffer(s) needed to perform BSCCLOSE, BSCOPEN, BSCREAD, and BSCWRITE operations.

If you are sending variable-length records, the length field (length1 operand) must contain the actual length of the message to be written. Reset the value coded for the length field to the buffer length before issuing a READ. Figure 4 on page LR-40 lists the number of buffers required for each type of BSCREAD and BSCWRITE operation.

Syntax:

```
label        BSCIOCB    lineaddr,buffer1,length1,buffer2,
                        length2,pollseq,pollsize,P1=,P2=,
                        P3=,P4=,P5=,P6=,P7=

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

Operand	*Description*
label	The label of the BSCIOCB. The BSCCLOSE, BSCOPEN, BSCREAD, and BSCWRITE instructions refer to this label. Other instructions can use the label to obtain additional status information stored in the first word of the BSCIOCB. After text is successfully received, this word contains the address of the last character received. For all other conditions, the word contains the Interrupt Status Word from the Series/1 BSC Adapter.
lineaddr	The hardware address, in hexadecimal, of the line on which the operation is to be performed.
buffer1	The label of the first buffer used in an I/O operation. This buffer is located in the target address space. The target address space is determined during a BSCOPEN operation and is defined in $TCBADS. This address space is used as the address space of the buffer until another BSCOPEN operation changes it.
length1	The length, in bytes, of the first buffer.
buffer2	The label of the second buffer used in an I/O operation. This buffer is located in the target address space as defined by $TCBADS.
length2	The length, in bytes, of the second buffer.
pollseq	The address of the poll or selection sequence to be used in a multipoint control line initial operation.

BSCIOCB - Specify BSC line address and buffers *(continued)*

pollsize The length, in bytes, of the poll or selection sequence.

The polling and selection sequences consist of one to seven characters followed by: ENQ,(Read or Write Initial)[1]. You can find specific sequences for a given device in the device component description manual. Generally, a 3-byte pollsize is sufficient for a sequence of address,address,ENQ[1] between Series/1 processors. The device type tributary determines the actual sequence.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Read type	Number of buffers	Write type	Number of buffers
C	1	C	1
D	0	CV	2
E	1	CVX	2
I	1	CX	1
P	1	CXB	1
Q	0	D	0
R	1	E	0
U	1	EX	0
		I	1
		IV	2
		IVX	2
		IX	1
		IXB	1
		Q	1
		N	0
		U	1
		UX	2

Figure 4. Required Buffers for BSCREAD and BSCWRITE

1 Commas are for readability only and are not part of the data stream.

BSCOPEN - Prepare a BSC line for use

The BSCOPEN instruction prepares a binary synchronous line for use by a task. The instruction acquires use of the BSC line and prepares it for a subsequent read or write operation.

If the line is a switched manual line (TYPE=SM), BSCOPEN requests a Data Terminal Ready acknowledgement and waits for the telephone connection to be established. If the line is a switched auto-answer line (TYPE=SA), BSCOPEN waits indefinitely for the ring interrupt and then requests a Data Terminal Ready acknowledgement.

Note: BSCOPEN assumes that point-to-point lines have Data Terminal Ready (DTR) permanently set on.

Syntax:

```
label          BSCOPEN    bsciocb,ERROR=,X21RN=,P1=,P2=,P3=

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

Operand	***Description***
bsciocb	The label or indexed location of the BSCIOCB statement associated with the open operation.
ERROR=	The label of the instruction to be executed if an error occurs while opening the line. If you do not code this operand, control passes to the next sequential instruction. In either case, the return code reflects the results of the operation.
X21RN=	The label of the data area containing the name of a member in the X.21 Circuit Switched Network Support connection data set. This member contains the connection information for this BSCOPEN. See "X21RN Coding Example" on page LR-42 for the layout of the data area. This parameter must be coded for auto-call (TYPE=SE or TYPE=SM) if the default data set name is not used. This parameter is optional for direct call (TYPE=DC) and is ignored for all other connection types. (The default name and the data set contents are explained in the *Communications Guide*.)
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

BSCOPEN - Prepare a BSC line for use *(continued)*

X21RN Coding Example

The following example shows how to code the data area referred to by the X21RN operand. This data area contains the name of the X.21 Circuit Switched Network connection record data set. The data area must be eight characters long. If the data set name is less than eight characters, the remaining positions in the data area must contain blanks. (See the *Communications Guide* for additional information about the connection data set.)

```
          .
          .
          .
          BSCOPEN     BSCIOCB,X21RN=MYDS
          .
          .
          .
MYDS      DC          CL8'X21RNDS '     DATA SET NAME
```

Return Codes

The following are the return codes for X.21 Circuit Switched Network. All other BSC instruction return codes are listed with the BSCWRITE instruction under "Return Codes" on page LR-54.

BSCOPEN - Prepare a BSC line for use *(continued)*

Return Code	Condition
-32	System is unable to find X.21 support. Re-IPL the system.
-31	Not enough storage available to handle the number of X.21 requests. Use the $DISKUT2 SS command to allocate more storage for $X21. You can issue three simultaneous requests for every 256 bytes of storage allocated.
-30	Your supervisor does not contain X.21 support.
-29	System does not have enough storage available to load the X.21 support or the connection record data set, $$X21DS, is not on the IPL volume.
-27	Unrecoverable hardware error. If $LOG is active, check the error log record for the X.21 device for more details.
-25	Connection failed
-24	Time expired for the completion of a call request. Call request failed.
-23	You cancelled a call request with a $C command.
-22	Call request failed due to Public Data Network problems. Call progress signals invalid.
-21	Call request failed due to Public Data Network problems. Call progress signals incomplete.
-20	Call request failed and network would not allow the request to be retried. If $LOG is active, check the error log record for the X.21 device for more details.
-19	Number of retries exhausted for the call request. If $LOG is active, check the error log record for the X.21 device for more details.
-18	Hardware error for the 2080 feature card. I/O request could not be completed.
-16	The Network information field of the X.21 connection record has no plus sign or just a plus sign.
-15	The value in the Retry or Delay field of the X.21 connection record exceeds the maximum value allowed.
-14	The Retry or Delay field of the X.21 connection record contains a negative value.
-13	A comma must separate the Retry, Delay, and Network information fields of an X.21 connection record.
-12	The Retry or Delay field of the X.21 connection record contains an invalid character.
-11	System does not have enough storage to execute a call request.
-10	Not enough storage in partition 1 for X.21 to execute a request.
-9	An EDL instruction failed. If $LOG is active, check the error log record for the X.21 device to find the failing instruction.
16	Your supervisor does not contain X.21 support.
17	The connection type you defined on the BSCLINE statement is not valid for the X.21 Circuit Switched Network.
18	The 2080 feature card is incorrectly jumpered for use with the X.21 Circuit Switched Network.
19	The X.21 network has been deactivated (DCE CLEAR).
26	Registration or cancellation request processed
27	Redirection activated
28	Redirection deactivated

BSCREAD - Read data from a BSC line

The BSCREAD instruction reads data from a binary synchronous line. If the read operation is successful, the first word of the associated BSCIOCB contains the address of the last character read.

Syntax:

```
label          BSCREAD   type,bsciocb,ERROR=,END=,CHAIN=,
                         TIMEOUT=,P1=,P2=,P3=

 Required:     type,bsciocb
 Defaults:     CHAIN=NO,TIMEOUT=YES
 Indexable:    bsciocb
```

Operand	***Description***
type	The type of read operation you want to perform. The read operations listed below are described in detail under "BSCREAD Types" on page LR-45.

C	Read Continue
D	Read Delay
E	Read End
I	Read Initial
P	Read Poll
Q	Read Inquiry
R	Read Repeat
U	Read User

bsciocb	The label or indexed location of the BSCIOCB statement associated with the read operation.
ERROR=	The label of the instruction to be executed if an error occurs (return codes 10 through 99). If you do not code this operand, control passes to the next sequential instruction. In either case, the return code reflects the results of the operation.
END=	The label of the instruction to be executed if an ending condition occurs (return codes 1 through 6). If you do not code this operand, control passes to the next sequential instruction. In either case, the return code reflects the results of the operation.

BSCREAD - Read data from a BSC line *(continued)*

CHAIN= YES, to cause a write operation to take place before the read operation. Code CHAIN=YES for Read Poll (type P) and Read User (type U). The system chains the DCB for the read operation to the DCB for the write operation.

You must provide the address of the data for the write operation in the buffer2 field of the BSCIOCB instruction. This buffer is located in the target address space as defined by $TCBADS during a BSCOPEN operation. You also must define the length (in bytes) of the data for the write operation in the length2 field of the BSCIOCB.

Your program receives an error return code if the address of the data or the length of the data for the write operation is zero. No write or read operation is performed.

NO, to cause the read operation to take place before any write operation.

Note: You can code CHAIN=YES to respond to a POLL with an EOT and then immediately set up the next read poll operation. This may be necessary in direct-connect environments where the Series/1 is a tributary to an extremely fast polling device.

TIMEOUT= YES, to cause a time-out error to occur if the access method does not receive data within three seconds during a receive operation. The access method attempts to recover from the error the number of times that you coded on the RETRIES operand of the the BSCLINE statement that defines this line. In a Read Initial operation, a time-out can occur both when attempting to establish the correct initial sequence and during the subsequent read of the first record.

NO, to prevent a time-out error from occurring if the access method does not receive data within three seconds during a receive operation.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All BSC instruction return codes are listed with the BSCWRITE instruction under "Return Codes" on page LR-54.

BSCREAD Types

Type	*Operation*
C	**Read Continue** - Reads subsequent blocks of data after an initial block has been received with a Read Initial.
D	**Read Delay** - Acknowledges that a block of data was correctly received and asks the transmitting station to wait before sending the next block. You can issue several Read Delays before resuming transmission of data with a Read Continue.

BSCREAD - Read data from a BSC line *(continued)*

E **Read End** - Acknowledges that a block of data was correctly received and asks the transmitting station to stop sending data. You should issue only one Read End during a single transmission. Once you issue the Read End, issue Read Continues until you actually receive an EOT.

I **Read Initial** - Reads the first block of data in a transmission. After a successful Read Initial operation, issue Read Continues until you receive an EOT.

For a point-to-point operation (TYPE=PT,SA,SM), Read Initial monitors the line for an ENQ sent by the transmitting station, writes a positive response (ACK-0), and reads the message block that follows.

In a multipoint controller operation (TYPE=MC), Read Initial polls a tributary station and, if the response to polling is positive, reads the message text.

For a multipoint tributary operation (TYPE=MT), Read Initial writes a positive response (ACK-0) and reads the message block that follows.

P **Read Poll** - Reads the poll or select sequence received when the Series/1 is acting as a tributary station on a multipoint line (TYPE=MT). If the operation is successful, the specified buffer contains the sequence received starting with the second station (control unit) address character. The access method does not check the contents of the received data stream, including control characters.

Once it is polled or selected, your program should check the next operation requested and issue the appropriate Read/Write Initial operation.

If you code CHAIN=YES, you can provide data to be transmitted by a write operation before the Read Poll operation. For example, you can provide three synchronization (SYN) characters and an EOT to be transmitted before the Read Poll operation.

Q **Read Inquiry** - Reads an ENQ character. Read Inquiry returns an invalid sequence error if ENQ or EOT is not received. If EOT is received, the access method takes the END= exit, if specified.

R **Read Repeat** - Requests that the last block of data be retransmitted following an unsuccessful read operation.

The RETRIES operand on the BSCLINE statement determines the number of times the read operation attempts to recover from a common error condition. You can use Read Repeat, however, to attempt further recovery depending on the actual error encountered.

U **Read User** - Receives data without issuing a response. The access method does not check the data or attempt any error recovery.

If you code CHAIN=YES, you can provide data to be transmitted by a write operation before the Read User operation.

BSCREAD - Read data from a BSC line *(continued)*

Return Codes

All BSC instruction return codes are listed with the BSCWRITE instruction under "Return Codes" on page LR-54.

BSCWRITE - Write data to a BSC line

The BSCWRITE instruction writes data to a binary synchronous line.

Syntax:

```
label         BSCWRITE  type,bsciocb,ERROR=,END=,CHECK=,
                        P1=,P2=,P3=

 Required:    type,bsciocb
 Defaults:    CHECK=YES
 Indexable:   bsciocb
```

Operand	*Description*
type	The type of write operation you want to perform. The write operations listed below are described in detail under "BSCWRITE Types" on page LR-49.

C	Write Continue
CV	Write Continue Conversational
CVX	Write Continue Conversational Transparent
CX	Write Continue Transparent
CXB	Write Continue Transparent Block
D	Write Delay
E	Write End
EX	Write End Transparent
I	Write Initial
IV	Write Initial Conversational
IVX	Write Initial Conversational Transparent
IX	Write Initial Transparent
IXB	Write Initial Transparent Block
Q	Write Inquiry
N	Write NAK (negative acknowledgement)

BSCWRITE - Write data to a BSC line *(continued)*

U Write User

UX Write User Transparent

bsciocb The label or indexed location of the BSCIOCB statement associated with the write operation.

ERROR= The label of the instruction to be executed if an error occurs (return codes 10 through 99). If you do not code the operand, control passes to the next sequential instruction. In either case, the return code reflects the results of the operation.

END= The label of the instruction to be executed if an ending condition occurs (return codes 1 through 6). If you do not code this operand, control passes to the next sequential instruction. In either case, the return code reflects the results.

CHECK= YES, to allow normal checking of the response to occur. This parameter is only valid for type CV or CVX operations.

NO, to prevent the response from being checked for protocol validity. CHECK=NO provides a chained write-to-read operation similar to Write User and Read User.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

BSCWRITE Types

Type ***Operation***

C **Write Continue** - Writes subsequent blocks of data after an initial block has been written with a Write Initial operation.

Write Continue writes the message text and reads a response from the receiving station.

CV **Write Continue Conversational** - Writes subsequent blocks of data after an initial block has been written in conversational mode.

Write Continue Conversational writes the message text and reads a response into your buffer. The access method checks acknowledgement sequences and attempts error recovery when necessary. If text is received, a -2 return code is returned instead of the normal -1.

CVX **Write Continue Conversational Transparent** - Writes subsequent blocks of transparent data after an initial block has been written in conversational mode.

Write Continue Conversational Transparent writes the message text and the ending characters DLE ETX. It then reads a response into your buffer. The access method

BSCWRITE - Write data to a BSC line *(continued)*

checks acknowledgement sequences and attempts error recovery when necessary. If text is received, a -2 return code is returned instead of the normal -1.

CX **Write Continue Transparent** - Writes subsequent blocks of transparent data after an initial block has been written.

Write Continue Transparent writes the message text and the ending characters DLE ETX. The operation then reads a response from the receiving station.

CXB **Write Continue Transparent Block** - Writes subsequent blocks of transparent data after an initial block has been written. This operation is the same as BSCWRITE type CX except that it uses ETB as the ending character instead of ETX.

Write Continue Transparent Block writes the message text and the ending characters DLE ETB. It then reads a response from the receiving station.

D **Write Delay** - Informs the remote station that the transmission of the next block of data will be delayed. You can perform several Write Delay operations before data transmission resumes.

Write Delay writes a temporary text delay (TTD) to the receiving station and reads a NAK response. The purpose of this operation is to inform the receiving station of a TTD before data transmission resumes.

E **Write End** - Informs the remote station that the previous block of data completed the write operation. Write End writes an EOT.

EX **Write End Transparent** - Writes a transparent EOT (DLE EOT). You can use this operation to notify the receiving station on a switched line that the transmitting station is disconnecting from the line.

I **Write Initial** - Writes the first block of data in a transmission. Write Initial establishes the correct initial sequence (depending on the type of line), writes the first block, and checks the response.

For a point-to-point operation (TYPE=PT,SA,SM), Write Initial:

- Writes an ENQ to gain use of the line
- Reads a positive response (ACK-O)
- Writes the message text
- Reads the response to the message text.

In a multipoint controller operation (TYPE=MC), Write Initial:

- Selects a tributary station

BSCWRITE - Write data to a BSC line *(continued)*

- Waits for a positive response to the selection
- Writes the message text
- Reads the response to the message text.

For a multipoint tributary operation (TYPE=MT), Write Initial:

- Writes the message text
- Reads a response from the controller station.

IV **Write Initial Conversational** - Writes the first block of data for a transmission in conversational mode.

Write Initial Conversational establishes the correct initial sequence (depending on the type of line), writes the first block of the message text, and reads a response into your buffer. The access method checks acknowledgement sequences and attempts error recovery when necessary. If text is received, a -2 return code is returned instead of the normal -1.

For a point-to-point operation (TYPE=PT,SA,SM), Write Initial Conversational:

- Writes an ENQ to gain use of the line
- Reads a positive response (ACK-O)
- Writes the message text
- Reads the response to the message text.

In a multipoint controller operation (TYPE=MC), Write Initial:

- Selects a tributary station
- Waits for a positive response to the selection
- Writes the message text
- Reads the response to the message text.

For a multipoint tributary operation (TYPE=MT), Write Initial:

- Writes the message text
- Reads a response from the controller station.

BSCWRITE - Write data to a BSC line *(continued)*

IVX **Write Initial Conversational Transparent** - Writes the first block of transparent data of a transmission in conversational mode.

Write Initial Conversational Transparent establishes the correct initial sequence (depending on the type of line), writes the first block of the message text and the ending characters DLE ETX. It then reads a response into your buffer. The access method checks acknowledgement sequences and attempts error recovery when indicated. If text is received, a -2 return code is returned instead of the normal -1.

For point-to-point operation (TYPE=PT,SA,SM): Write Initial Conversational Transparent:

- Writes an ENQ to gain use of the line
- Reads a positive response (ACK-O)
- Writes the message text
- Writes the required ending characters DLE ETX
- Reads the response to the message text.

In a multipoint controller operation (TYPE=MC), Write Initial:

- Selects a tributary station
- Waits for a positive response to the selection
- Writes the message text
- Writes the required ending characters DLE ETX
- Reads the response to the message text.

For a multipoint tributary operation (TYPE=MT), Write Initial:

- Writes the message text
- Writes the required ending characters DLE ETX
- Reads a response from the controller station.

IX **Write Initial Transparent** - Writes the first block of transparent data in a transmission. Write Initial Transparent establishes the correct initial sequence (depending on the type of line), writes the first block of transparent data, and checks the response. The access method terminates the message text with DLE ETX.

BSCWRITE - Write data to a BSC line *(continued)*

IXB **Write Initial Transparent Block** - Same as Write Initial Transparent (IX) except that ETB is used as the ending character instead of ETX.

Q **Write Inquiry** - Writes an ENQ character and reads the response into your buffer. The response is either a control sequence or text.

Use this operation to request that a response to a message block be retransmitted. The access method retries the operation if it times out.

N **Write NAK** - Writes a NAK (negative acknowledgement) character. Use this operation to respond "device not ready" to polling or selection when the Series/1 operates as a tributary station on a multipoint line (TYPE=MT).

U **Write User** - Transmits a character stream. The access method does not perform an associated read operation or attempt error recovery.

UX **Write User Transparent** - Transmits a transparent character stream. The access method does not perform an associated read operation or attempt error recovery.

The operation concludes with one of the following character pairs contained in BSCIOCB buffer2: DLE ETX, DLE ETB, or DLE ENQ.

BSCWRITE - Write data to a BSC line *(continued)*

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Return Code	Condition
-2	Text received in conversational mode
-1	Successful completion
END=	
1	EOT received
2	DLE EOT received
3	Reverse interrupt received
4	Forward abort received
5	Remote station not ready (NAK received)
6	Remote station busy (WACK received)
ERROR=	
10	Time-out occurred
11	Unrecovered transmission error (BCC error)
12	Invalid sequence received
13	Invalid multi-point tributary write attempt
14	Disregard this block sequence received
15	Remote station busy (WACK received)
20	Wrong length record - long (No COD)
21	Wrong length record - short (write only)
22	Invalid buffer address
23	Buffer length zero
24	Undefined line address
25	Line not opened by calling task
30	Modem interface error
31	Hardware overrun
32	Hardware error
33	Unexpected ring interrupt
34	Invalid interrupt during auto-answer attempt
35	Enable or disable DTR error
99	Access method error

BUFFER - Define a storage area

The BUFFER statement defines a data storage area. The standard buffer contains an index word, a length word, and a data buffer.

The index word indicates the number of bytes stored in the buffer, but only when incremented by your program. A label assigned to the index word in your program will enable you to increment and reset the index word from the program. The system sets the index word to 0 when it creates the buffer. The length word indicates the total length of the buffer in bytes.

Certain instructions, for example INTIME and SBIO allow you to add new entries sequentially to a buffer by referring to and incrementing the index word.

You can use a BUFFER statement to define the storage area needed for use with the Host Communications Facility TP READ/WRITE instruction. The use of the BUFFER statement to set up a temporary I/O buffer for a terminal is explained under the IOCB statement.

READTEXT and GETEDIT instructions may be used to modify the BUFFER statement. PRINTEXT and PUTEDIT instructions use the BUFFER statement to determine the number of values to print.

Figure 5 on page LR-57 shows the physical layout of a buffer.

Syntax:

```
label         BUFFER   length,item,INDEX=

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

Operand	*Description*
length	The length of the buffer in terms of the data item (words or bytes) you specify. The system allocates two words of control information, the index word and the length word, in addition to the buffer itself. The length must not exceed 16,380 words or 32,760 bytes. If your program includes a READ instruction that will use the buffer, the buffer area should be a multiple of 256 bytes. **Note:** When filling a buffer, you should be careful not to exceed the buffer size. The system does not check for an overflow condition.

BUFFER - Define a storage area *(continued)*

item 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 BYTE or BYTES if you are using the BUFFER statement with a CALL $IMOPEN instruction.

Code TPBSC to generate a buffer for use with the TP READ/WRITE instruction (Host Communications Facility). The count operand reflects the length of the buffer in bytes when you code TPBSC.

INDEX= The label of the buffer index word. Do not code this operand if you coded TPBSC for the item operand. You can think of this operand as a pointer to the next available data location in the buffer.

BUFFER - Define a storage area *(continued)*

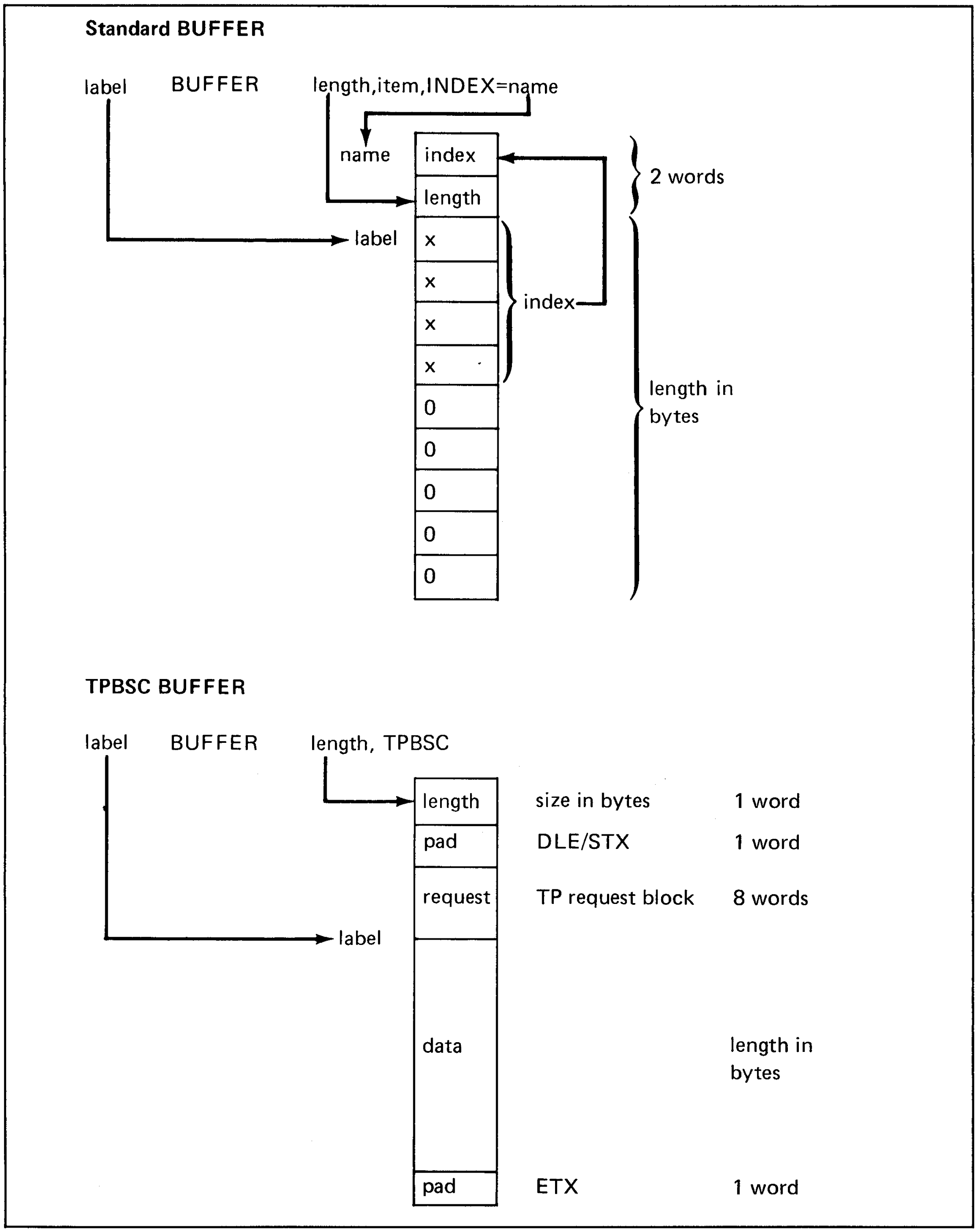

Figure 5. Physical Layout of a Buffer

BUFFER - Define a storage area *(continued)*

Coding Example

The BUFFER statement labeled BUFF defines a 102-word storage area. The first word of this area is labeled INDX as coded on the keyword INDEX. The second word contains the count of the total number of BUFFER entries. The remaining 100 words are the actual BUFFER storage area.

```
                .
                .
                .
        SUBROUT    STORE
        IF         (INDX,GE,198)
          ENQT       $SYSPRTR
          PRINTEXT   '@BUFFER IS FULL'
          DEQT
          RETURN
        ENDIF
        MOVEA      #1,BUFF                 MOVE ADDR OF BUFF
        ADD        #1,INDX                 INCREMENT #1
        MOVE       (0,#1),DATA1,(1,WORD)   MOVE DATA TO BUFF
        ADD        INDX,2                  INCREMENT BUFFER INDEX
        RETURN
BUFF    BUFFER     100,WORDS,INDEX=INDX
DATA1   DATA       F'0'
```

CACLOSE - Close a Channel Attach port

The CACLOSE instruction terminates the connection between your application program and a Channel Attach port and disables the port from receiving interrupts from the System/370.

Syntax:

```
label          CACLOSE    caiocb,ERROR=,P1=

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

Operand	*Description*
caiocb	The label or indexed location of the Channel Attach I/O control block defined for this port.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CACLOSE and your program must test for errors before issuing a WAIT.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Syntax Examples

1) The following example closes a port defined by the CAIOCB at USERIOCB.

```
CLOSE10    CACLOSE    USERIOCB
```

2) This example closes a port defined by the CAIOCB at the indexed location of USER plus the contents of #1. If an error occurs, the instruction at label E1 receives control.

```
CLOSEFC    CACLOSE    (USER,#1),ERROR=E1
```

CACLOSE - Close a Channel Attach port *(continued)*

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CACLOSE post codes are returned to the first word of of the CAIOCB you defined for the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
	FE0C	-500	Data pending from host
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		EXIO error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
524	020C		Timeout
	0234	564	Users CAIOCB not linked to port
567	0237	567	System error; CAPGM terminating
	0238	568	Port not opened

Channel attach codes 501-513 are the same as the EXIO post codes 1-13 respectively.

CAIOCB - Create a Channel Attach port I/O control block

The CAIOCB statement creates a Channel Attach port I/O control block that contains the information your program requires to use a port.

You supply the device address, the port number, and the label of the first buffer control area. You must provide a CAIOCB for all operations to a port. Do not try to modify the CAIOCB during program execution.

Syntax:

```
label        CAIOCB     address,PORT=,BUFFER=

 Required:   label,address,PORT=,BUFFER=
 Defaults:   none
 Indexable:  none
```

Operand	***Description***
label	The label of the CAIOCB for use with the CAOPEN, CACLOSE, CAREAD, and CAWRITE instructions.
address	A two-digit hexadecimal device address.
PORT=	The number of the port (0-31) for which this I/O control block is being created.
BUFFER=	The label of a three-word area containing:

- First word - the address of the buffer to be used for the first read.
- Second word - the number of bytes to be used.
- Third word - the partition number of the buffer. If this word is zero, the system assumes the buffer is in the partition in which you loaded your program.

Syntax Example

The following statement creates a Channel Attach port I/O control block for port 3. The device address is 10.

```
USERIOCB  CAIOCB    10,PORT=3,BUFFER=AREA
```

CALL - Call a subroutine

The CALL instruction executes a system subroutine or a subroutine that you write. You can pass up to five parameters as arguments to the subroutine. If the subroutine you call is a separate object module to be link-edited with your program, you must code an EXTRN statement with the subroutine name in the calling program. Figure 6 on page LR-64 shows an example of a primary task calling a subroutine which in turn calls a second subroutine.

Syntax:

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

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

Operand ***Description***

name The name of the subroutine to be executed.

par(n) The parameters you want to pass to the subroutine. You can pass up to five single-precision integers or the labels of single-precision integers or null parameters to the subroutine. The CALL instruction replaces the parameters specified in the subroutine with the parameters you specify. For example, the instruction replaces the first parameter of the subroutine with par1, the second parameter with par2, and so on.

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

If the parameter to be passed is the label of an EQU instruction, you can code a plus sign (+) in front of that label. The plus sign causes the value equated to the label to be passed to the subroutine. If you do not code a plus sign in front of the label, the instruction assumes that the value equated to the label is an address and passes the data at that address as the parameter.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CALL - Call a subroutine *(continued)*

Syntax Examples

1) Call the PROG subroutine and pass it a value of 5.

```
CALL    PROG,5
```

2) Call the PROG subroutine and pass it a value of 5 and the null parameter 0.

```
CALL    PROG,5,
```

3) Call the SUBROUT subroutine and pass it the contents of PARM1, the address of PARM2, and the value of the equated label FIVE.

```
CALL    SUBROUT,PARM1,(PARM2),+FIVE
```

Coding Example

The following coding example shows a use of the CALL instruction. The main routine calls the subroutine READREC. A relative record number is passed to the subroutine as RECNUMBR and is received as RECORD#.

Two methods of passing an address to a subroutine are illustrated. First, at label MA, the address of ENDFILE is moved to EOF. Then EOF is passed to the subroutine as a parameter of a CALL instruction.

Second, in the same CALL instruction, the address of READERR is passed to the subroutine by enclosing the label in parentheses. When EOF and READERR are passed to the subroutine, they are referred to as EOFEXIT and ERREXIT, respectively.

The EOFEXIT and ERREXIT parameters are addresses. In order to branch to the locations these parameters represent, they must be enclosed in parentheses as the object of a GOTO instruction.

The subroutine uses the relative record number defined by RECORD# to read the data file. An end-of-file condition causes a branch to the appropriate exception routine whose address is contained in EOFEXIT.

A read error will cause a branch to the location whose address is contained in ERREXIT. If no exception condition is encountered, control is returned to the calling routine by the RETURN instruction.

CALL - Call a subroutine *(continued)*

```
             .
             .
             .
MA        MOVEA    EOF,ENDFILE
          CALL     READREC,RECNUMBR,EOF,(READERR)
          GOTO     CONTINU
READERR   EQU      *
          PRINTEXT '@ ERROR ENCOUNTERED READING DISK FILE RECORD NUMBER'
          PRINTNUM RECNUMBR
          PROGSTOP
ENDFILE   EQU      *
          PRINTEXT '@ END OF INPUT DATA FILE REACHED'
          PROGSTOP
CONTINU   EQU      *
            .
            .
            .
          SUBROUT  READREC,RECORD#,EOFEXIT,ERREXIT
          READ     DS1,DISKBUFR,1,RECORD#,END=ENDEXIT,ERROR=ERRORXIT
          RETURN
ENDEXIT   EQU      *
          GOTO     (EOFEXIT)
ERRORXIT  EQU      *
          GOTO     (ERREXIT)
            .
            .
            .
```

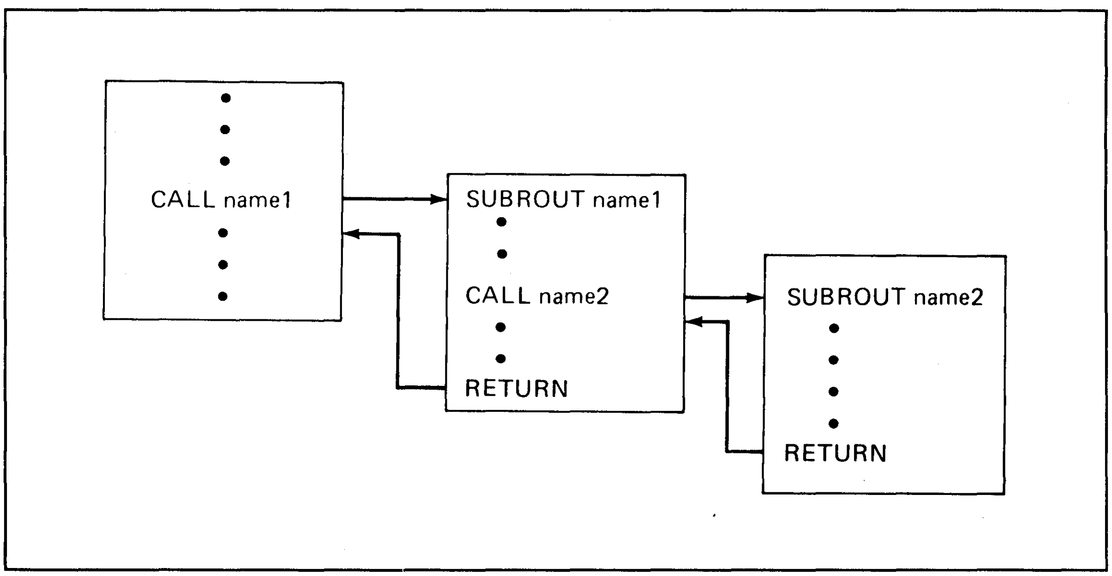

Figure 6. Execution of Subroutines

CALLFORT - Call a FORTRAN subroutine or program

The CALLFORT instruction calls a FORTRAN program or subroutine from an Event Driven Executive program. If you call a FORTRAN main program, the name you specify for the name operand is the name you coded on the FORTRAN PROGRAM statement or the default name, MAIN, if no PROGRAM statement was coded. If you call a FORTRAN subroutine, specify the name of the subroutine for the name operand. You can pass parameters to FORTRAN subroutines. Standard FORTRAN subroutine conventions apply to the use of CALLFORT.

If separate tasks within an EDL program each contain CALLFORT instructions, the tasks should not execute concurrently because the FORTRAN subroutines are serially reusable and not reentrant.

For a more complete description of the use of the CALLFORT instruction, see the *IBM Series/1 Event Driven Executive FORTRAN IV Program 5719-FO2 User's Guide*, SC34-0315.

Syntax:

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

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

Operand	*Description*
name	The name of a FORTRAN program or subroutine, consisting of 1 to 6 alphameric characters, that begins with an alphabetic character. You must also code this name, or entry point, on an EXTRN statement.
a1,a2,an	A list of parameters or arguments (a1,a2, and so on) that you want to pass to the subroutine. The argument can be a constant, a variable, or the name of a buffer. If you are passing the subroutine only one argument, you do not have to enclose it in parentheses.
p1,p2,pn	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands. Each name in this list can be up to eight characters long. The system assigns the first name in the list to the first argument, the second name in the list to the second argument, and so on.

CALLFORT - Call a FORTRAN subroutine or program *(continued)*

Syntax Examples

1) Call the SORT1 subroutine.

```
SAMPLE   PROGRAM    START
         EXTRN      SORT1
START    EQU        *
         CALLFORT   SORT1
```

2) Call the SUM subroutine and pass it an integer constant of 5.

```
SAMPLE   PROGRAM    START
         EXTRN      SUM
START    EQU        *
         CALLFORT   SUM,5
```

3) Call the SUM subroutine and pass it variables A and B.

```
SAMPLE   PROGRAM    START
         EXTRN      SUM
START    EQU        *
         CALLFORT   SUM,(A,B)
           .
           .
           .
A        DATA       F'5'
B        DATA       F'0'
```

4) Call the SUM subroutine and pass it variables A and B. Assign the label INPUT to argument A and OUTPUT to argument B.

```
SAMPLE   PROGRAM    START
         EXTRN      SUM
START    EQU        *
         CALLFORT   SUM,(A,B),P=(INPUT,OUTPUT)
           .
           .
           .
A        DATA       F'5'
B        DATA       2F'0'
```

CAOPEN - Open a Channel Attach port

The CAOPEN instruction establishes a connection between your application program and a Channel Attach device port.

You must issue a CAOPEN instruction before your program can use a port for data transfer. When your program opens a Channel Attach port, it has exclusive use of the port until the port is closed. The system rejects any request to open a port already opened.

Syntax:

```
label          CAOPEN      caiocb,ERROR=,P1=

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

Operand	***Description***
caiocb	The label or indexed location of the Channel Attach port I/O control block you defined for this port.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CAOPEN and your program must test for errors before issuing a WAIT.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Syntax Examples

1) Open a port defined by the CAIOCB at label USERIOCB.

```
OPEN10     CAOPEN     USERIOCB
```

2) Open a port defined by the CAIOCB at the indexed location of USER plus the contents of #1. If an error occurs, the instruction at label E1 receives control.

```
OPENFC     CAOPEN     (USER,#1),ERROR=E1
```

CAOPEN - Open a Channel Attach port *(continued)*

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CAOPEN post codes are returned to the first word of of the CAIOCB you defined for the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		EXIO error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
520	0208		Interrupt error
524	020C		Timeout
	0227	551	Device not started
	0228	552	Stop in progress
	022C	556	Port out of range
	022D	557	Port already open
	022E	558	Read buffer not provided
	022F	559	Read buffer count = 0
567	0237	567	System error; CAPGM terminating
	023A	570	Device in diagnostic mode

Channel attach codes 501-513 are the same as the EXIO post codes 1-13, respectively.

CAPRINT - Print Channel Attach trace data

The CAPRINT instruction prints the entire trace area on your printer or terminal. Use this instruction for problem determination. Tracing is disabled while printing is being done.

Syntax:

```
label        CAPRINT    address,event,TITLE=,CONSOLE=,ERROR=,
                        P1=,P2=,P3=,P4=

Required:    address
Defaults:    CONSOLE=$SYSPRTR
Indexable:   EVENT,TITLE
```

Operand	*Description*
address	A two-digit hexadecimal device address.
event	The label or indexed location of the event to be posted when printing has completed. If you do not code this operand, your program is not posted when printing completes.
TITLE=	The label or indexed location of a two-word area defining the title on the trace data listing. The first word contains the address of the title. The second word contains the length, in bytes, of the title. If you do not code this operand, no title appears on the trace data listing. TITLE= cannot exceed 72 bytes if you are using the $CHANUT1 utility.
CONSOLE=	The label of the IOCB statement that defines the terminal used as the output device for this trace print request.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CAPRINT and your program must test for errors before issuing a WAIT.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CAPRINT - Print Channel Attach trace data *(continued)*

Syntax Examples

1) Print trace data for the device at address 10 on $SYSPRTR.

```
PRINT10    CAPRINT    10,ERROR=ERROR2
```

2) Print trace data for the device at address FC on PRTR2. When the printing completes, the instruction posts the event at the indexed location of address A plus the contents of #1.

```
PRINTFC     CAPRINT    FC,(A,#1),TITLE=HEAD,                               X
                       CONSOLE=PRTR2,ERROR=E1
```

Return Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

For detailed explanations of the return codes, refer to *Messages and Codes*.

Hex	Return Code	Explanation
0227	551	Device not started
0228	552	Stop in progress
022A	554	Device not found

CAREAD - Read from a Channel Attach port

The CAREAD instruction reads data from a Channel Attach port. The operation occurs at the port you specify in the CAIOCB statement.

Syntax:

```
label         CAREAD     caiocb,thisbuf,nextbuf,ERROR=,
                         P1=,P2=,P3=

Required:     caiocb,thisbuf,nextbuf
Defaults:     none
Indexable:    caiocb,thisbuf,nextbuf
```

Operand	***Description***
caiocb	The label or indexed location of the Channel Attach port I/O control block defined for this port.
thisbuf	The label of a three-word area containing: • First word - the address of the buffer receiving the data from this read • Second word - the number of bytes to be read into the buffer • Third word - the partition number of the buffer
nextbuf	The label of a three-word area containing: • First word - the address of the buffer to be used for the next read • Second word - the number of bytes to be read into the buffer • Third word - the partition number of the buffer
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CAREAD, and your program must test for errors before issuing a WAIT.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CAREAD - Read from a Channel Attach port *(continued)*

Syntax Examples

1) Read data from the port defined by the CAIOCB at label USERIOCB. The address of the buffer receiving the data is in the 3-word area at label BUF1.

```
READ10     CAREAD     USERIOCB,BUF1,BUF2
```

2) Read data from the port defined by the CAIOCB at the indexed location of USER plus the contents of #1. The address of the buffer receiving the data is in the 3-word area at the indexed location of BUF1 plus the contents of #2.

```
READFC     CAREAD    (USER,#1),(BUF1,#2),                                    X
                   (BUF2,#1),ERROR=E1
```

CAREAD - Read from a Channel Attach port *(continued)*

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CAREAD post codes are returned to the first word of the CAIOCB you defined for the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		EXIO error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
524	020C		Timeout
520	0208		Interrupt error
521	0209		Negative acknowledgement (write only)
522	020A		Buffer overlay (read only)
523	020B		Protocol error
	022E	558	Buffer not provided
	022F	559	Buffer count = 0
	0232	562	Write buffer not provided
	0233	563	Write buffer count = 0
	0234	564	Users CAIOCB not linked to port
567	0237	567	System error; CAPGM terminating
	0238	568	Port not opened

Channel attach codes 501-513 are the same as the EXIO post codes 1-13, respectively.

CASTART - Start Channel Attach device

The CASTART instruction starts a Channel Attach device. Your program must start the Channel Attach device before it can open any of the device's ports.

The first CASTART instruction you issue loads the Channel Attach device handler program, initializes the control blocks for the device, and prepares the device to accept interrupts from the System/370. Subsequent CASTART instructions connect to the device handler program initially loaded.

Syntax:

```
label         CASTART    address,ecb,ERROR=,P1=,P2=

Required:     address,ecb
Defaults:     none
Indexable:    ecb
```

Operand	*Description*
address	A two-digit hexadecimal device address.
ecb	The label or indexed location of the event to be posted upon completion of the CASTART operation.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CASTART, and the program must test for errors before issuing a WAIT.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

The CASTART instruction in the following example starts the device at address 10. When the start operation ends, the instruction posts the event at $ECB.

```
START10    CASTART    10,$ECB
```

CASTART - Start Channel Attach device *(continued)*

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CASTART post codes are returned to the first word of of the event control block (ECB) you defined in the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		EXIO error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
524	020C		Timeout
525	0200		Not a Channel Attach device
	0228	552	Stop in progress
	&22A	554	Device not found
567	0237	567	System error; CAPGM terminating
	0239	569	Device already started

Channel Attach codes 501-513 are the same as the EXIO post codes 1-13, respectively.

CASTOP - Stop a Channel Attach device

The CASTOP instruction stops a Channel Attach device and disables the device from receiving interrupts from the System/370. Your program can stop a device only if no ports are open. When your program stops the last device, the Channel Attach device handler program terminates.

Syntax:

```
label         CASTOP     address,ecb,ERROR=,P1=,P2=

Required:     address,ecb
Defaults:     none
Indexable:    ecb
```

Operand	*Description*
address	A two-digit hexadecimal device address.
ecb	The label or indexed location of the event to be posted upon completion of the CASTOP operation.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CASTOP, and your program must test for errors before issuing a WAIT.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CASTOP - Stop a Channel Attach device *(continued)*

Syntax Example

The CASTOP instruction in the following example stops the device at address 10. When the operation ends, the instruction posts the event at $ECB.

```
STOP10     CASTOP     10,$ECB
```

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CASTOP post codes are returned to the first word of of the event control block (ECB) you defined in the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
524	020C		Timeout
	0227	551	Device not started
	0228	552	Stop in progress
	0229	553	Device in use
	022A	554	Device not found
567	0237	567	System error; CAPGM terminating
	023A	570	Device in diagnostic mode
599	0257		$CAPGM has ended

Channel attach codes 501-513 are the same as the EXIO post codes 1-13, respectively.

CATRACE

CATRACE - Control Channel Attach tracing

The CATRACE instruction controls the collection of I/O trace data for a Channel Attach device. You can turn tracing on or off.

This instruction collects Channel Attach trace data in processor storage which can slow system performance. For this reason, you should use the CATRACE instruction primarily for problem determination.

Syntax:

```
label        CATRACE    address,ENABLE=,ERROR=,P1=

Required:    address
Defaults:    ENABLE=YES
Indexable:   none
```

Operand	*Description*
address	A two-digit hexadecimal device address.
ENABLE=	YES (the default), to turn on or enable tracing. NO, to turn off or disable tracing.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CATRACE and your program must test for errors.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Syntax Examples

1) Turn on tracing for the device at address 10.

```
TRACE10    CATRACE    10
```

2) Turn off tracing for the device at address FC. If an error occurs, the instruction at label E1 receives control.

```
TRACEFC    CATRACE    FC,ENABLE=NO,ERROR=E1
```

CATRACE - Control Channel Attach tracing *(continued)*

Return Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

For detailed explanations of the return codes, refer to *Messages and Codes*.

Hex	Return Code	Explanation
0227	551	Device not started
0228	552	Stop in progress
022A	554	Device not found
0235	565	Trace already on
0238	566	Trace already off

CAWRITE - Write to a Channel Attach port

The CAWRITE instruction sends data to a Channel Attach port. The operation occurs at the port you specify in the CAIOCB statement.

Syntax:

```
label          CAWRITE    caiocb,buffer,ERROR=,P1=,P2=

Required:      caiocb,buffer
Defaults:      none
Indexable:     caiocb,buffer
```

Operand	*Description*
caiocb	The label or indexed location of the Channel Attach port I/O control block defined for this port.
buffer	The label of a three-word area containing: • First word - the address of the buffer containing the data to be sent. • Second word - the number of bytes to be sent. • Third word - the partition number of the buffer. If this word is zero, the system assumes the buffer is in the partition in which you loaded your program.
ERROR=	The label of the instruction to be executed if an error occurs. If you do not code this operand, control passes to the next instruction after the CAWRITE, and your program must test for errors before issuing a WAIT.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) Write data to a port defined by the CAIOCB at label USERIOCB. BUFA is the label of the 3-word area that contains the address of the buffer from which the data is to be sent.

```
WRITE10    CAWRITE    USERIOCB,BUFA
```

2) Write data to a port defined by the CAIOCB at a location specified in #1. The address of the buffer containing the data to be sent is specified in a 3-word area located at an address in #2.

```
WRITEFC    CAWRITE    #1,#2,ERROR=ERROR1
```

CAWRITE - Write to a Channel Attach port *(continued)*

Return and Post Codes

Return codes are returned in the first word of the task control block of the program or task issuing the instruction. A return code other than -1 indicates that the link module found an error before the instruction performed an I/O operation. Your program must check the return code before it issues a WAIT because a WAIT should only be used if an I/O operation is being performed.

CAWRITE post codes are returned to the first word of of the CAIOCB you defined for the instruction.

For detailed explanations of the return and post codes, refer to *Messages and Codes*.

Post Code	Hex	Return Code	Explanation
-1	FFFF	-1	Successful
501	01F5		EXIO error-device not attached
502	01F6		EXIO error-busy
503	01F7		EXIO error-busy after reset
504	01F8		EXIO error-command reject
505	01F9		EXIO error-intervention required
506	01FA		EXIO error-interface data check
507	01FB		EXIO error-controller busy
508	01FC		EXIO error-channel command not allowed
509	01FD		EXIO error-no DDB found
510	01FE		EXIO error-too many DCBs chained
511	01FF		EXIO error-no residual status address
512	0200		EXIO error-zero bytes specified for residual status
513	0201		EXIO error-broken DCB chain
516	0204		EXIO error-device already opened
520	0208		Interrupt error
521	0209		Negative acknowledgement (write only)
522	020A		Buffer overlay (read only)
523	020B		Protocol error
524	020C		Timeout
	022E	558	Buffer not provided
	022F	559	Buffer count = 0
	0232	562	Write buffer not provided
	0233	563	Write buffer count = 0
	0234	564	Users CAIOCB not linked to port
567	0237	567	System error; CAPGM terminating
	0238	568	Port not opened

Channel attach codes 501-513 are the same as the EXIO post codes 1-13, respectively.

COMP - Define location of message text

The COMP statement points to a data set or module that contains formatted program messages. The MESSAGE, READTEXT, GETVALUE, and QUESTION instructions refer to the label of the COMP statement when retrieving program messages.

The COMP statement also assigns a four-character prefix to the messages your program obtains. This prefix, the number of the message being retrieved, and the message text are the components that make up a complete program message.

You must code at least one COMP statement in a program that retrieves program messages. The message utility, $MSGUT1, formats the messages you write for your programs. Refer to the *Operator Commands and Utilities Reference* for a description of this utility. See Appendix E, "Creating, Storing, and Retrieving Program Messages" on page LR-615 for more information.

Syntax:

```
label         COMP     'idxx',name,TYPE=

 Required:    label,'idxx',name
 Defaults:    TYPE=STG
 Indexable:   none
```

Operand	***Description***
label	The label you specified for the COMP= keyword on a MESSAGE, READTEXT, GETVALUE, or QUESTION instruction.
'idxx'	A four-character prefix that identifies the messages your program obtains through this COMP statement. The system displays this prefix with the message text when you code MSGID=YES on a MESSAGE, READTEXT, GETVALUE or QUESTION instruction.
name	The name of the module or data set that contains the formatted messages. For a module, this is the name you assigned to the module with the STG option of the message utility, $MSGUT1. This name can be up to eight characters long. **Note:** You must link-edit the message module with your program. For a disk or diskette data set, specify the name in the form DSx, where "x" indicates the position of the message data set in the list of data sets you defined on the PROGRAM statement. DS1, for example, refers to the first data set in the list. DS2 refers to the second data set in the list, and so on. The valid range for "x" is from 1 to 9.

COMP - Define location of message text *(continued)*

If your program contains a DSCB instruction, you can use the label you coded on the DS#= operand for this operand.

TYPE= STG (the default), if the messages reside in a module that you link-edit with your program.

DSK, if the messages reside in a disk or diskette data set.

Syntax Examples

1) The COMP statement in this example points to the message module PROMPTS. The MESSAGE instruction, which retrieves the first message in PROMPTS, refers to the label of the COMP statement. Because the MESSAGE instruction contains MSGID=YES, the system displays the prefix PROM and the number of the message before the message text.

```
         MESSAGE   1,COMP=A,SKIP=1,MSGID=YES
          .
          .
          .
         PROGSTOP
A        COMP      'PROM',PROMPTS,TYPE=STG
```

2) The COMP statement in this example points to the message data set MESSAGE1 on volume EDX002. The GETVALUE instruction, which retrieves the fifth message from MESSAGE1, refers to label of the COMP statement.

```
MESSAGE  PROGRAM   START,DS=(MESSAGE1,EDX002)
          .
          .
          .
         GETVALUE  INPUT,5,SKIP=1,COMP=B
         PROGSTOP
B        COMP      'MSG1',DS1,TYPE=DSK
```

CONCAT - Concatenate two character strings

The CONCAT instruction concatenates two character strings, or a character string and a graphic-control character. The instruction places the contents of string2 to the right of any contents in string1. The resulting character string remains in string1.

CONCAT changes the character count of string1 after the operation to reflect the original contents of string1 plus the concatenated data from string2. Truncation on the right occurs if the combined counts exceed the physical length of string1.

Note: To use the CONCAT statement, you must specify an AUTOCALL to $AUTO,ASMLIB during program preparation (link-edit.)

Syntax:

```
label          CONCAT     string1,string2,RESET,REPEAT=,P1=,P2=

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

Operand	*Description*
string1	The label of a data string to which the contents of string2 are concatenated.
string2	The data to be concatenated to string1. You can code the label of a character string, a one-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	Resets the character count of string1 to zero before starting the CONCAT operation. The count is not reset if you omit this operand.
REPEAT=	The number of times string2 is to be concatenated to string1. For example, if string2 contains C' ' and you code REPEAT=5, five blanks are concatenated to the contents of string1. Code a positive integer for this operand.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CONCAT - Concatenate two character strings *(continued)*

Syntax Examples

1) Concatenate ESC to TEXT1. Reset the character count of TEXT1 before the operation.

```
CONCAT   TEXT1,ESC,RESET
```

2) Concatenate the control character FF to TEXT1.

```
CONCAT   TEXT1,FF
```

CONTROL

CONTROL - Perform tape operations

The CONTROL instruction 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 tapemarks, rewind the tape, erase the tape, set the tape drive offline, or rewind the tape and set the tape drive offline. With the 4968 tape unit, the CONTROL instruction allows you to write at a density of 1600 bits per inch or 3200 bits per inch.

In addition, you can use the CONTROL instruction to close tape data sets. You should close all tape data sets. If you do not close data sets, you must control the tape drive directly with the various CONTROL functions.

When you close an SL (standard-label) output tape, the CONTROL instruction writes the following trailer label: TM EOF1 TM TM. The instruction writes the following label when you close an NL (nonlabeled) tape: TM TM.

Input tapes are automatically rewound as the result of a close operation. An attempt to write a tapemark to an unexpired file is an error condition.

If you have two tape drives on one controller and they receive concurrent rewind requests, one tape drive waits for the other to complete. To allow concurrent rewinds to multiple standard label tape drives on one controller, you must issue the "CONTROL DSxx,REW" instruction to each open tape drive.

Syntax:

```
label         CONTROL  DSx,type,count,END=,ERROR=,WAIT=,P1=,P3=

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

Operand	***Description***
DSx	The data set you want to use. Code DSx, where "x" is the relative number of the data set in the list of data sets you defined on the PROGRAM statement. DS1, for example, points to the first data set in the list; DS2 points to the second data set, and so on. You can substitute a DSCB name defined by a DSCB statement for this operand.
type	The CONTROL function to be performed. The following functions are available:

FSF Forward space file (tapemark). Regardless of where the tape is currently positioned, the tape searches forward the number of tape marks indicated in the count operand. If the specified number of

CONTROL - Perform tape operations *(continued)*

tapemarks indicated by the count field is not on the tape, the positioning of the tape is unpredictable.

BSF Backward space file (tapemark). The tape searches backward until the next tapemark is read. The default value for count is 1. If the tape is at load point when your program issues this command, 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 spaces 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 your program issues this command, the load point return code is returned.

WTM Write tapemark. This function writes a tapemark on the tape. If the count field is coded, successive tapemarks are 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). For standard-label tapes, the tape is repositioned to the HDR1 label of the data set. For nonlabeled tapes, the tape is positioned to the beginning of the first data record. You can use $VARYON to change the file number being processed or you can use a CONTROL function.

Once you close a tape data set, you must call DSOPEN to open the data set before you can use it again. You can call DSOPEN with the CALL instruction or invoke the subroutine implicitly by having the name of the data set in another program header.

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

DEN16 Sets the density of the 4968 tape unit to 1600 bits per inch. This function is not valid for other tape devices.

To set the density, the tape must be at the load point.

CONTROL - Perform tape operations *(continued)*

DEN32 Sets the density of the 4968 tape unit to 3200 bits per inch. This function is not valid for other tape devices.

To set the density, the tape must be at the load point.

ERASE Erases forward from the point where the tape is positioned to a point five feet beyond the end-of-tape marker (EOT). The function then rewinds the tape and unloads it.

The system sends out a device interrupt when the tape is at the load point and ready.

count The number of files or records to be skipped or the number of tapemarks to be written. You can code a constant or the label of a count value.

END= The label of the first instruction of the routine to be invoked if the system detects an "end-of-data-set" (EOD) condition (return code=10). If you do not specify this operand, the system treats an EOD as an error. Do not code this operand if you code WAIT=NO.

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

ERROR= The label of the first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not specify this operand, control passes to the next sequential instruction in your program and you must test the return code in the first word of the task control block for errors. Do not code this operand if you code WAIT=NO.

WAIT= If WAIT is not coded, 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, and any other option will be ignored.

For functions other than close, if the operand is coded as WAIT=NO, control is 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 first word of the task control block (TCB) (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.

CONTROL - Perform tape operations *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) The instruction closes the tape data set specified by DS1, rewinds the tape, and sets the tape drive offline.

```
CONTROL    DS1,CLSOFF
```

2) The instruction causes the tape data set specified by DS2 to be spaced forward 16 data records.

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

CONTROL - Perform tape operations *(continued)*

Coding Example

The following program uses the CONTROL FSF command, at label C1, to advance the "master name file" to the third data set on a nonlabeled tape. The program asks the operator if he or she wants to search the file for a particular name. If the answer is 'yes', the program requests the file name.

At label C2, a CONTROL FSR command advances the tape file to record 90. If the end-of-file is reached before the tape is positioned to the target record, control passes to an error routine (not shown).

The program then reads a record and compares the name field in it to the name the operator entered. This sequence continues until the program finds the name the operator entered or until the end-of-file is reached.

Assuming the program finds the name, it prints the name (and accompanying file information) and the record for the names before and after it.

If the name is the first on the file (INDEX=1), the program can only print the name and the record that immediately follows it. Therefore, the CONTROL BSR command, at label C3, uses the P3= parameter naming operand to determine dynamically how many records to back space. The count is 1, if the name is in the first data record on the file, or 2, if the name is not in the first data record on the file.

A DO loop at label LOOP2 reads the name records and prints them. If the end-of-file is reached before the last record can be printed, the program passes control to an error routine (not shown).

At label C4, the tape is backspaced past the tapemark preceding the name file and at label C5, the tape is positioned to the first record on the file. Control then passes to the beginning of the program.

CONTROL - Perform tape operations *(continued)*

```
FILESRCH   PROGRAM    START,DS=(NAMEFILE,TAPE01)
START      EQU        *
C1         CONTROL    DS1,FSF,3,ERROR=DS1ERROR
INQUIRE    EQU        *
           QUESTION   '@DO YOU WISH TO SEARCH THE MASTER NAME FILE ?',NO=END
           PRINTEXT   '@PRECEEDING AND SUCCEEDING NAMES WILL ALSO BE LISTED'
           READTEXT   NAME,'@ENTER SUBJECT NAME UP TO 12 CHARACTERS'
C2         CONTROL    DS1,FSR,90,END=DS1ENDF1,ERROR=DS1ERROR
           MOVE       INDEX,0
LOOP       EQU        *
           ADD        INDEX,1
           READ       DS1,BUFR,END=DS1ENDF2
           IF         (BUFR,NE,NAME,(12,BYTES))
           GOTO       LOOP
           ENDIF
           IF         (INDEX,LE,1)
             PRINTEXT '@NAME AT BEGINNING OF FILE - ONLY 2 LISTED'
             MOVE   COUNT,2
           ELSE
             MOVE   COUNT,3
             MOVE   INDEX,2
           ENDIF
C3         CONTROL    DS1,BSR,2,P3=INDEX
           DO         1,TIMES,P1=COUNT
           READ       DS1,BUFR,END=LASTONE
           MOVE       BUFR,TEXT,(50,BYTES)
           PRINTEXT   TEXT,SKIP=1
           ENDDO
C4         CONTROL    DS1,BSF
C5         CONTROL    DS1,FSF
           GOTO       INQUIRE
*************************************
           DATA       X'3232'
TEXT       DATA       50C' '
NAME       TEXT       LENGTH=12
DS1ENDV    EQU        *
           .
           .
           .
DS1ERROR   EQU        *
           .
           .
           .
```

CONTROL - Perform tape operations *(continued)*

Tape Return Codes and Post Codes

Tape return codes are returned in the first word of the task control block of the program that issues the instruction.

Return Code	Condition
-1	Successful completion.
1	Exception but no status.
2	Error reading cycle steal status.
3	I/O error; retry count exhausted.
4	Error issuing READ CYCLE STEAL STATUS.
6	I/O error issuing I/O operations.
10	End of data; a tape mark was read.
21	Wrong length record.
22	Device not ready.
23	File protected.
24	End of tape.
25	Load point.
26	Unrecoverable I/O error.
27	SL data set not expired.
28	Invalid blocksize.
29	Offline, in-use, or not open.
30	Incorrect device type.
31	Close incorrect address.
32	Block count error during close.
33	Close detected on EOV1.

The following post codes are returned to the event control block (ECB) of the calling program.

Post Code	Condition
-1	Function successful.
101	TAPEID not found.
102	Device not offline.
103	Unexpired data set on tape.
104	Cannot initialize BLP tapes.

CONVTB - Convert numeric string to EBCDIC

The CONVTB instruction converts both integer and floating-point values to an EBCDIC character string. You can also convert floating-point values to E notation.

Syntax:

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

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

Operand	*Description*
opnd1	The label of a storage area where the converted results are to be placed. The system stores the results beginning at the label referred to by this operand. The converted results are in EBCDIC. Opnd1 must be a different storage location than opnd2.
opnd2	The label of a storage area containing the value to be converted to EBCDIC. You must know the form (precision) 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 form of opnd2. The valid precisions 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 after the system converts it: **w** Width of the EBCDIC field in bytes. If the field will contain a decimal point or sign character (+ or -), include this in the count. **d** Number of digits to the right of the decimal point. This is valid for floating-point variables only. Code a 0 for integer values.

CONVTB - Convert numeric string to EBCDIC *(continued)*

t — Type of EBCDIC Data. Code I for integer data, F for floating-point data (XXXX.XXX), or E for a number in exponent (E) notation. See the value operand under the DATA/DC statement for a description of E notation format.

Px= — Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Notes:

1. Conversion routines assume that the type of variable to be converted is specified by the PREC operand. If the PREC operand is not specified, and if the variable is not of the default precision, incorrect results can occur.

2. Exponent (E) notation should be used for floating-point numbers greater than 10^{12}. Otherwise, a conversion error will occur.

Syntax Examples

1) The CONVTB instruction in the following example uses an integer value.

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

The value 12345 in the variable VALUE is converted to EBCDIC at TEXTA in the following format (b represents a blank):

bbb12345

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

2) In this example, the CONVTB instruction uses floating-point values.

```
          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
```

CONVTB - Convert numeric string to EBCDIC *(continued)*

The result of the CONVTB operation is (b represents a blank):

TEXTB=bbbbb62421.1600

TEXT1=b.49261392916000Eb07

Coding Example

This example demonstrates one use of the CONVTB instruction.

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

If, for example, the initial value of BNUMEXP is X'0038', the value of BMANHRS is X'431B0C00', and the value of BAVERAGE is X'4087915E8CA84482', the results of the program would appear as follows:

```
NUMBER OF EXPERIMENTS CONDUCTED : 56

TOTAL MAN-HOURS EXPENDED ON PROJECT : 432.75

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

CONVTB - Convert numeric string to EBCDIC *(continued)*

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
3	Conversion error

CONVTD - Convert EBCDIC string to numeric string

The CONVTD instruction converts an EBCDIC character string to an integer or floating-point numeric string.

Syntax:

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

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

Operand	***Description***
opnd1	The label of a storage area where the converted results are to be placed. Opnd1 must be a different storage location than opnd2. Make sure that you reserve enough space 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	A label that points to the first character of the EBCDIC character string. You can code the following range of data values:

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

*Valid range is from 10^{-85} through 10^{75}

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

CONVTD - Convert EBCDIC string to numeric string *(continued)*

- 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 the system finds any other character during the conversion, it takes the following action:

- If the delimiters , or / are found within a string:

  The system stops the conversion and returns a "successful completion" code (-1). Opnd1 contains the data the system converted before it found the delimiter.

- If the delimiter , or / or * or . is the first character found in a string:

  The system returns a "field omitted" code (2). The variable you defined in opnd1 (the target field) remains unchanged.

- If all blanks are found in opnd2:

  The system places zeros in opnd1 and returns a "successful completion" code (-1).

- If any other character (for example, an alphabetic character) is found within a string:

  The system returns a code of 1, "invalid data encountered during conversion." Data converted before the system found the invalid character is stored in opnd1.

- If only an invalid character is found in opnd2 or the value being converted is too large or too small:

  The system returns a "conversion error" (3). The contents of the variable you defined for opnd1 (the target field) are unknown.

CONVTD - Convert EBCDIC string to numeric string *(continued)*

The following table shows the results of several conversion operations using the default format (6,0,I):

Input	Return Code	Output
12	-1	12
12,	-1	12
12/	-1	12
(blanks)	-1	0
12C	1	12
12.B	1	12
12 C	1	12
,	2	(target field unchanged)
/	2	(target field unchanged)
*	2	(target field unchanged)
.	2	(target field unchanged)
A	3	(target field unchanged)
1234567	3	(value of target field unknown)

PREC= The form of opnd1. The valid precisions 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 to be converted:

w Width of the EBCDIC field in bytes. If the field will contain a decimal point or sign character (+ or -), include this in the count.

d Number of digits to the right of the decimal point. This option is valid only for floating-point variables. Code a 0 for integer values.

t Type of EBCDIC Data. Code I for integer data, F for floating-point data (XXXX.XXX), or E for a number in exponent (E) notation. See the value operand under the DATA/DC statement for a description of E notation format.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

CONVTD - Convert EBCDIC string to numeric string *(continued)*

Syntax Examples

1) The following CONVTD instruction uses an integer value.

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

Note: 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.

2) The CONVTD instruction in this example uses floating-point values.

```
         CONVTD   VALUE,TEXT1,PREC=F,FORMAT=(5,1,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
```

Note: 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.

Coding Example

The following example demonstrates one use of the CONVTD instruction:

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

CONVTD - Convert EBCDIC string to numeric string *(continued)*

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

opnd1	**Before**	**After**
BUNIT	X'0000'	X'1999'
BMILES	X'00000000'	X'42298AFB'
BRESPONS	X'0000000000000000'	X'422A73F2D016AE42'

opnd2	**Before**	**After**
UNIT	6553bb	X'F6F5F5F34040'
MILES	41.5429bbb	X'F4F14BF5F4F2F9404040'
RESPONSE	42.45292378bbbb	X'F4F24BF4F5F2F9F2F3F7F840404040'

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

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

COPY - Copy source code into your source program

The COPY statement copies source code into your source program. The operation occurs each time you compile or assemble the program containing the COPY statement.

The source code you copy must be in a disk or diskette data set. The source code must not contain a COPY statement. The system copies the source code into your source program immediately following the COPY statement.

To prevent the system from printing the source code in your listing each time you compile your program, code PRINT OFF before the COPY statement and PRINT ON following it. See the program example given in "PRINT - Control printing of a compiler listing" on page LR-321 for more detail.

Syntax:

```
blank         COPY     name

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

Operand ***Description***

name The name of the data set on disk or diskette that contains the source code to be copied into your source program.

Notes:

1. When using the $EDXASM compiler, if the source code to be copied is not on volume ASMLIB, you must code a *COPYCOD statement in the $EDXL data set to indicate on what volume the source code resides. $EDXL is on volume ASMLIB. Refer to the *Customization Guide* for an explanation of the *COPYCOD statement.
2. For details on using the COPY statement with the Series/1 macro assembler, refer to IBM Series/1 Event Driven Executive Macro Assembler (5719-ASA).
3. For details on using the COPY statement with the System/370 macro assembler, refer to the *IBM System/370 Program Preparation Facility*, SB30-1072.

System Equates

This section contains the equate names for some commonly used system control blocks. Coding the COPY statement with the equate name gives you a listing of the control block. You can use the equates in the control block listing to refer to and obtain data from fields within the control block. When you compile programs with the host or Series/1 macro assemblers, the system

COPY - Copy source code into your source program *(continued)*

includes the following equate names in your program when it encounters a PROGRAM statement: PROGEQU, TCBEQU, DDBEQU, CMDEQU, and DSCBEQU.

The *Internal Design* contains a complete list of the control blocks in the system. The control block equates reside on volume ASMLIB and end with the characters "EQU".

BSCEQU Provides a map of the control block built by the BSCLINE system definition statement.

Note: BSCEQU is also the name of a macro in the macro libraries that the host and Series/1 macro assemblers use. Do not attempt to copy BSCEQU when using either of the macro assemblers.

CCBEQU Provides a map of the control block (CCB) built by the TERMINAL system definition statement.

CMDEQU Provides a map of the supervisor's emulator command table built by the PROGRAM statement.

DDBEQU Provides a map of the device data block (DDB) built by the DISK system definition statement.

DDODEFEQ Provides a table that defines the format of disk directory control entries (DCEs) and member entries.

COPY - Copy source code into your source program *(continued)*

DSCBEQU	Provides a map of the data set control block (DSCB) built by the PROGRAM or DSCB statements.
ERRORDEF	Provides equates for use in checking the return codes from the LOAD, READ, WRITE, and SBIO instructions.
FCBEQU	Provides a map of an Indexed Access Method file control block (FCB) for use with the EXTRACT subroutine.
IAMEQU	Provides a set of symbolic parameter values for use in constructing parameter lists for calls to Indexed Access Method subroutines.
PROGEQU	Provides maps of the program header, built by the PROGRAM statement, and the supervisor's communication vector table (CVT).
TCBEQU	Provides a map of the task control block (TCB) built by the TASK or PROGRAM statements.
STOREQU	Provides a map of the storage control block built by the STORBLK statement.

COPY - Copy source code into your source program *(continued)*

Coding Example

The following example uses a COPY statement to copy the source code labeled CHKBUFR into a source program.

```
          .
          .
          CALL      CHKBUFR,BUFRSIZE,(EOBUFFER)
          .
          .
          COPY      CHKBUFR
          .
          .
```

When the source program is compiled, the COPY statement copies the following code into the source program:

```
          SUBROUT   CHKBUFR,BUFFLEN,BUFFEND
          SUBTRACT  BUFFLEN,1
          IF        (BUFFLEN,GE,MAX)
          GOTO      (BUFFEND)
          ENDIF
          ADD       BUFFLEN,1
          RETURN
          .
          .
MAX       DATA      F'256'
          .
          .
          .
```

CSECT - Identify object module segments

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

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 instructions. However, all CSECTs, after the first one, generate ENTRY instead of CSECT definitions.

Program modules assembled by 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
```

Operand	***Description***
label	The label must be the name of the program module for the first CSECT. For following CSECTs the label must be an entry name.

CSECT - Identify object module segments *(continued)*

Coding Example

In module A, the first CSECT statement signifies that the program can be entered at label GETTIME. In module B, the CSECT statement defines label GOTTIME as being an entry point. The ENTRY statement in module A will allow the time to be printed without the "TIME IS NOW" text.

MODULE A

```
              .
              .
GETTIME       CSECT
              ENTRY     GETTIME2
              EXTRN     GOTTIME
              .
              .
GETTIME       EQU       *
              PRINTEXT  '@THE TIME IS NOW'
GETTIME2      EQU       *
              PRINTIME
              GOTO      GOTTIME
              .
              .
              .
```

MODULE B

```
              .
              .
GOTTIME       CSECT
              EXTRN     GETTIME
              .
              .
TIME          EQU       *
              GOTO      GETTIME
GOTTIME       EQU       *
              .
              .
              .
```

DATA/DC - Define data

The DATA/DC statement defines the data you are using in your program. You can represent data in the following forms: binary, integer, hexadecimal, character, floating-point, or address.

Within a single DATA statement, you can define one or more character strings or variables. With programs you compile under $EDXASM, you can code up to 10 separate data specifications on a single DATA statement by separating the individual specifications with commas. When you assemble programs under $S1ASM, a DATA statement can contain only one data specification.

Syntax:

```
label        DATA     dup type value
label        DC       dup type value

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

Operand	*Description*
dup	Duplication factor for the data type you define.
type	Data type or form of data representation. The valid data types are:

Code	Data type	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	Integer, signed fullword	2 bytes
H	Integer, signed halfword	1 byte
D	Integer, signed doubleword	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

value The value to be assigned to the data area. This operand is also the field length for some data types. The value is enclosed in quotes for all data types except A, in which the value is enclosed in parentheses.

Notes:

1. Except for A-type data (address), the value must be a self-defining term and cannot be defined with an EQU statement.

DATA/DC - Define data *(continued)*

2. The maximum number of hexadecimal digits you can specify for this operand is 8; the maximum number of characters you can specify is 15.

3. For programs compiled under $EDXASM, the value operand can define a maximum of 65,535 bytes.

Considerations when Defining Data

The allowable ranges for data values are:

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

*Valid range is from 10^{-85} to 10^{75}

You can express floating-point values as real numbers with decimal points (for example 1.234) or in exponent (E) notation. E notation uses the form:

SX.XXESYY

where:

S = Optional sign character (+ or -); default is (+)
X = Characteristic of 1 to 6 numeric digits for PREC=E, or 15 digits for PREC=L
. = 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)

When coding character strings (C), you can specify a field length by coding the type as CLn, where "n" is the length of the field in bytes. If the length of the the character string you specify is less than the field length chosen, the balance of the field to the right of the string is filled with blanks. To specify the field length for hexadecimal values (X), code the type as XLn. If the length of the hexadecimal value you specify is less than the field length chosen, the balance of the field to the left of the value is filled with zeros.

Neither $EDXASM nor $S1ASM support such complex data expressions as:

```
DATA  A(B-C)
```

where B is an external label.

DATA/DC - Define data *(continued)*

Syntax Examples

The following examples show some of the ways that you can define data in your program.

1) Hexadecimal 30F in binary. This format is not allowed with $EDXASM.

```
BINCON    DATA    B'001100001111'
```

2) An integer constant of 1.

```
A         DATA    F'1'
```

3) 128 words of 0.

```
BUF       DC      128F'0'
```

4) The EBCDIC string 'XYZ'.

```
CHAR      DATA    C'XYZ'
```

5) 80 EBCDIC blanks.

```
BLANK     DC      80C' '
```

6) The character '$' followed by seven blanks.

```
C8        DC      CL8'$'
```

7) The integer 241 in hexadecimal

```
HEXV      DATA    X'00F1'
```

8) The address of 'BUF'.

```
ADDR      DATA    A(BUF)
```

9) The 2-word integer constant 100,000

```
DBL       DATA    D'100000'
```

DATA/DC - Define data *(continued)*

10) The floating-point value 1.234

```
F1        DATA    E'1.234'
```

11) Four floating-point values of 0.123 (4 bytes for each value).

```
F2        DATA    4E'0.123'
```

12) Four extended-precision floating-point values of 12345678.9 (8 bytes for each value).

```
L2        DATA    4L'12345678.9'
```

13) An extended-precision floating-point value in exponent (E) form.

```
L3        DATA    L'123456E-40'
```

14) A word with a value of 1 and a doubleword with a value of 2.

```
MANY      DATA    F'1',D'2'
```

15) The hexadecimal string X'0001'.

```
X         DC      XL2'1'
```

16) The hexadecimal string X'000123'.

```
Y         DC      XL3'123'
```

DCB - Create a device control block

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
```

Operand	*Description*
PCI=	YES, to cause the device to present a program-controlled interrupt at the completion of the DCB fetch before data transfer. NO (the default), does not cause the device to present a program-controlled interrupt.
IOTYPE=	INPUT, for operations involving transfer of data from device to processor or for bidirectional transfers under one DCB operation. OUTPUT (the default), for operations involving transfer of data from processor to device or for control operations involving no data transfer.
XD=	YES, if the DCB is a nonstandard type. NO (the default), if the DCB is a standard type.
SE=	YES, to allow the device to suppress the reporting of certain exception conditions. NO (the default), to report all exception conditions.
DEVMOD=	The byte that describes functions unique to a particular device. This byte is in word 0 of the device's DCB. 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 plus sign (+).
DVPARM2=	The value of device-dependent parameter word 2. Code as four hexadecimal digits or the label of an EQU preceded by a plus sign (+).

DCB - Create a device control block *(continued)*

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

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 from 0 to 32767 or the label of an EQU preceded by a plus sign (+).

DATADDR= The label of the first byte of data to be transferred.

For information on the contents of DVPARM1-DVPARM4 and DEVMOD, refer to the description manual of the device you are using.

DCB - Create a device control block *(continued)*

Syntax Examples

1) The DCB labeled WR1DCB is for an output operation in which the 120-byte field labeled MSG1 will be transferred to the device. IOTYPE= defaults to OUTPUT. The device places any status information from the operation in RESTAT.

```
WR1DCB   DCB   SE=YES,DVPARM1=0300,DVPARM2=3048,DVPARM3=1100,           X
               DVPARM4=RESTAT,CHAINAD=WR2DCB,COUNT=120,                 X
               DATADDR=MSG1
          .
          .
          .
MSG1     DATA  120X'00'
RESTAT   DATA  2F'0'
```

2) The DCB labeled WR2DCB is for a type of device-control operation. IOTYPE defaults to OUTPUT but no data transfer occurs because the statement does not contain the DATADDR or COUNT operands. The device places any status information from the operation in RESTAT.

```
WR2DCB   DCB   SE=YES,DVPARM1=20A0,DEVMOD=6F,DVPARM4=RESTAT
          .
          .
          .
RESTAT   DATA  2F'0'
```

Coding Example

For a coding example using a DCB statement, see the example following the description of the EXIO instruction.

DEFINEQ - Define a queue

The DEFINEQ statement defines the queue descriptor (QD) and a set of queue entries (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 the *Event Driven Executive Language Programming Guide.*

Syntax:

```
label         DEFINEQ   COUNT=,SIZE=

Required:     label, COUNT=
Defaults:     SIZE=2 (2 bytes of data for each element in the
              free queue chain)
Indexable:    none
```

Operand	***Description***
label	The label of the queue that this statement creates.
COUNT=	The number of 3-word queue entries (QEs) to be generated. The system also generates a 3-word queue descriptor (QD) and assigns the first word of the QD the label of the DEFINEQ statement. "Queue Layout" on page LR-116 describes the structure of a queue. The COUNT operand must be specified using a self-defining term; an equated value is not allowed. This operand must also be a positive number greater than 0.
SIZE=	The size, in bytes, of each buffer (data area) to be included in the buffer pool in the initial queue. The system generates as many buffers as you specified in the COUNT operand. It initializes each buffer to binary zeros. Each QE in the queue contains the address of an associated buffer in the buffer pool. If you do not specify the SIZE operand, the system places all QEs in the free chain and the queue is defined as empty. If you specify SIZE, the system includes all QEs in the active chain and the queue is defined as full.

DEFINEQ - Define a queue *(continued)*

Queue Layout

A queue is composed of a queue descriptor (QD) and one or more queue entries (QEs). Figure 7 on page LR-117 shows the layout of a queue.

The DEFINEQ statement generates a 3-word QD. Word 1 of the QD is a pointer to the most recent entry in a chain of active QEs. Word 2 is a pointer to the oldest entry in a chain of active QEs. Word 3 is a pointer to the first QE in a chain of free QEs. If the 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.

DEFINEQ also generates several 3-word QEs. Word 1 of the oldest QE in the active chain points back to the QD. For the rest of the QE's in the active chain, word 1 is a pointer to the next most recent QE in the chain.

Word 2 of the most recent QE in the active chain points back to the QD. For the rest of the QEs in the active chain, word 2 is a pointer to the next oldest QE in the chain.

Word 3 of a QE in the active chain is a queue entry. The entry is a 16-bit word that can be a data item or the address of an associated data buffer.

When a QE is in the free chain, word 3 is a pointer to the next element in the free chain. Word 3 of the last QE in the free chain is a pointer back to the QD.

DEFINEQ - Define a queue *(continued)*

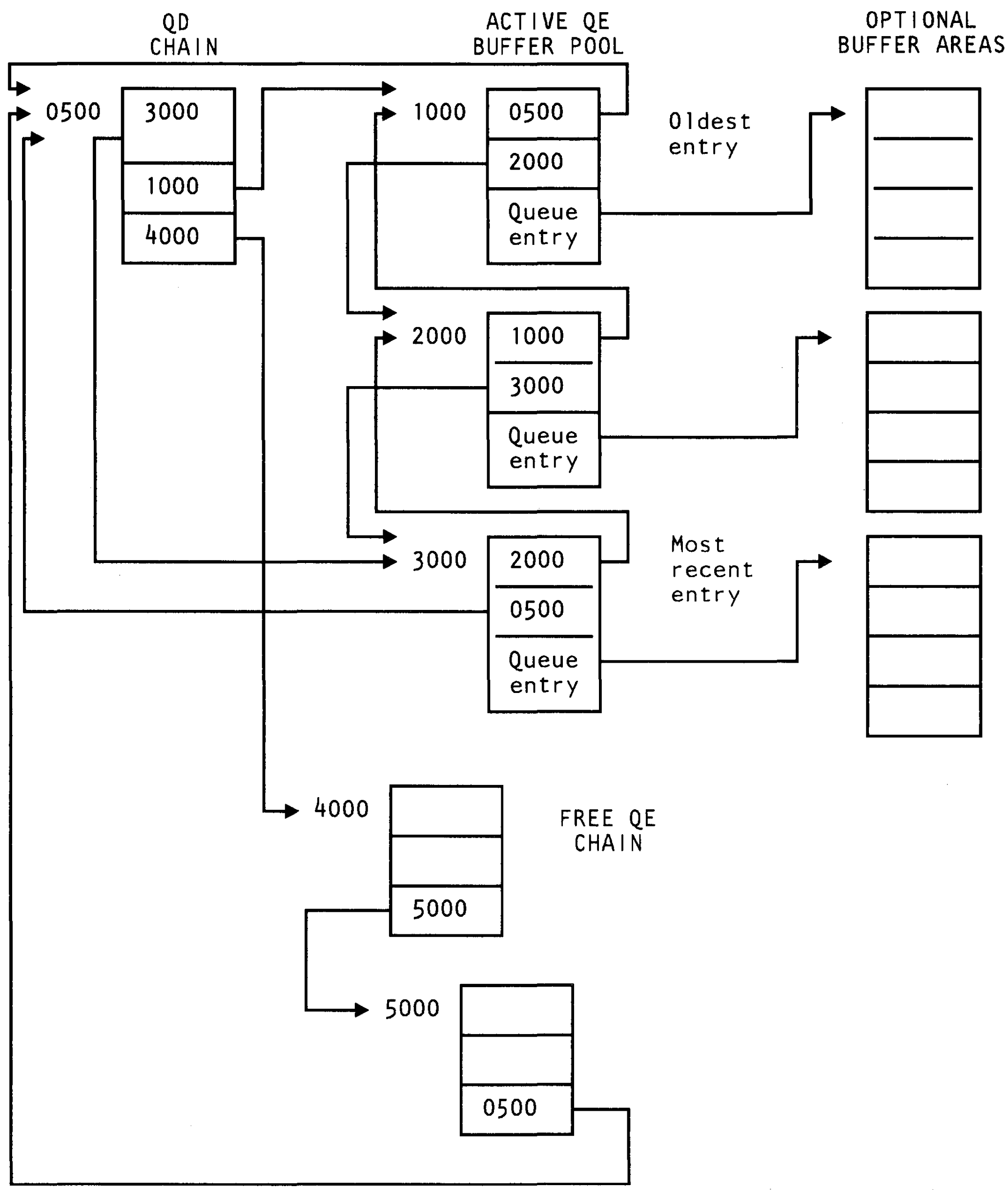

Figure 7. Layout of a Queue

DEFINEQ - Define a queue *(continued)*

Syntax Examples

1) The statement generates a 3-word queue descriptor (QD), followed by four 3-word queue entries (QE). All four of the QEs are placed in the QE free chain.

```
QUE1      DEFINEQ   COUNT=4
```

2) The statement generates a 3-word QD, followed by two 3-word QEs and two 6-word queue data areas (one 6-word area for each of the QEs) initialized to binary zeros. Because the SIZE operand is specified, all QEs are included in the active chain and the queue is defined as full.

```
QUE2      DEFINEQ   COUNT=2,SIZE=12
```

DEQ - Release a resource for use

The DEQ instruction releases exclusive control of a resource *other* than a terminal by releasing control of the queue control block (QCB) associated with that resource.

You acquire exclusive control of the QCB associated with a resource with the ENQ instruction. (See the ENQ instruction for more information.) Your program must release exclusive control of, or "dequeue," a QCB associated with a resource before other programs can use the resource again.

DEQ normally assumes that the QCB for the resource is defined in the same partition as the current program. However, your program can dequeue a QCB in another partition by using the cross-partition service capability of DEQ. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example that dequeues a resource in another partition. Refer to the *Event Driven Executive Language Programming Guide* for more information on cross-partition services.

When you use the $S1ASM macro assembler or the host assembler, the DEQ instruction causes the assembler to generate a QCB for a resource at the end of the program. When you use $EDXASM, no QCBs are generated; you must use the QCB statement to generate the QCBs your program requires.

Syntax:

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

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

Operand	***Description***
qcb	The label of the QCB to be dequeued. This must be the same label 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) associated with the resource. Your program can examine the code word by referring to the label of the QCB. A code of 0 is interpreted by the ENQ instruction to mean that the resource is unavailable for use; all non-zero codes show that the resource is available. You must code a self-defining term for this operand.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

See "ENQ - Gain exclusive control of a resource other than a terminal" on page LR-148 for an example using the DEQ instruction.

DEQT - Release a terminal for use

The DEQT instruction releases control of the terminal that your program acquired control of with an ENQT instruction.

When an ENQT instruction redefines the characteristics of a terminal through an IOCB statement, DEQT restores the terminal characteristics defined on the TERMINAL definition statement. (See *Installation and System Generation Guide* for information on the TERMINAL statement.) DEQT also causes partially full buffers to be written to the terminal, completes all pending I/O, and forces the cursor or forms to the next line (carriage return.) In addition, you can use the DEQT instruction to end spooling to a printer assigned to your program.

Your program also releases exclusive control of a terminal when it executes a PROGSTOP instruction.

The supervisor places a return code in the first word of the task control block (taskname) whenever a DEQT instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

When coding the DEQT instruction, you can include a comment which will appear with the instruction on your compiler listing. If you include a comment, you must also code the CLOSE operand. The comment must be separated from the operand field by at least one blank and it may not contain commas.

Syntax:

```
label          DEQT     CLOSE=  comment

Required:      none
Defaults:      CLOSE=NO
Indexable:     none
```

Operand	***Description***
CLOSE=	This operand provides additional control for spool jobs.

Code CLOSE=YES to logically end a spool job. Logically ending a SPOOL job allows the executing program to create separate printed output on the spool device. This operand has no effect on the DEQT instruction if the device to which the DEQT is directed is not a spool device, or if spool is not active.

Code CLOSE=ALL to end all spool jobs associated with this task and all other tasks in the program that have previously issued a DEQT instruction.

Coding CLOSE=NO (the default) has no affect on the DEQT instruction or spool operation.

DEQT - Release a terminal for use *(continued)*

Syntax Examples

1) Release control of the system printer, $SYSPRTR.

```
         ENQT    $SYSPRTR
           .
           .
         DEQT
```

2) Release control of the device TTY1.

```
         ENQT    TERM1,BUSY=ALTERN
           .
           .
         DEQT    CLOSE=NO        THIS IS A COMMENT
           .
           .
         PROGSTOP
TERM1    IOCB    TTY1,PAGSIZE=24
```

DETACH - Deactivate a task

The DETACH instruction removes a task from operational status. A task can only detach itself. If a program reattaches a task, execution begins with the instruction following the DETACH in the reattached task.

Syntax:

```
 label         DETACH    code,P1=

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

Operand	*Description*
code	The posting code to be inserted in the terminating ECB ($TCBEEC) of the task being detached. A complete list of TCB equates is in the *Internal Design*.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

DETACH - Deactivate a task *(continued)*

Coding Example

The following program announces the start of each race at a racetrack.

TASKA is the program's primary task. It starts, or "attaches," TASKB which enqueues the track announcement board at label RACEBORD (code not shown). TASKB then prints the time of day and the number of the race which is about to begin. When TASKB completes, it executes a DETACH instruction and detaches itself from the program.

When the primary task reattaches TASKB at label A2, the GOTO instruction immediately following the DETACH instruction executes. The GOTO instruction passes control back to the beginning of the TASKB and execution resumes at the label BEGIN.

```
TASKA     PROGRAM   START
START     EQU       *
            .
            .
          ATTACH    TASKB
            .
            .
A2        ATTACH    TASKB
            .
            .
          PROGSTOP
            .
            .
TASKB     TASK      BEGIN
BEGIN     EQU       *
          ENQT      RACEBORD
          ADD       NUMBER,1
          PRINTEXT  '@THE TIME IS NOW'
          PRINTIME
          PRINTEXT  ' AND RACE# '
          PRINTNUM  NUMBER
          PRINTEXT  ' OF THE DAY IS ABOUT TO BEGIN '
          DEQT
          DETACH
          GOTO      BEGIN
NUMBER    DATA      F'0'
          ENDTASK
          ENDPROG
          END
```

DIVIDE - Divide integer values

The DIVIDE instruction divides an integer value in operand 1 by an integer value in operand 2. The values can be positive or negative. To divide floating-point values, use the FDIVD instruction.

See the DATA/DC statement for a description of the various ways you can represent integer data.

The system stores the remainder of the operation (an integer) in the first word of the task control block (TCB). This remainder will be lost if a subsequent instruction issues a return code and updates the TCB. The remainder is double-precision only if operand 2 is double precision.

The system indicates an overflow for the DIVIDE operation by placing a X'80000000' in the first two words of the TCB. X'80000000' is also the result of a divide by zero operation.

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
```

Operand	***Description***
opnd1	The label of the data area containing the value divided by opnd2. Opnd1 cannot be a self-defining term. The system stores the result of the DIVIDE operation in opnd1 unless you code the RESULT operand.
opnd2	The value by which opnd1 is divided. You can specify a self-defining term or the label of a data area. The value of opnd2 does not change during the operation.
count	The number of consecutive values on which the system performs the operation. The maximum value is 32767.
RESULT=	The label of a data area or vector in which the result is placed. The data area you specify for opnd1 is not changed if you specify RESULT. This operand is optional.
PREC=xyz	Specify the precision of the operation in the form xyz, where x is the precision for opnd1, y is the precision for opnd2, and z is the precision of the result ("Mixed-precision Operations" on page LR-125 shows the precision combinations allowed for the DIVIDE instruction). You can specify single precision (S) or double precision (D) for each operand. Single precision is a word in length; double precision is two words in length. The default for opnd1, opnd2, and the result is single precision.

DIVIDE - Divide integer values *(continued)*

If you code a single letter for PREC, the letter applies to opnd1 and the result. Opnd2 defaults to single precision. If, for example, you code PREC=D, opnd1 and the result are double precision and opnd2 defaults to single precision.

If you code two letters for PREC, the first letter applies to opnd1 and the result, and the second letter applies to opnd2. With PREC=DD, for example, opnd1 and the result are double precision and opnd2 is double precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Mixed-precision Operations

The following table lists the precision combinations allowed for the DIVIDE instruction:

opnd1	opnd2	Result	Precision	Remarks
S	S	S	S	default
S	S	D	SSD	-
D	S	D	D	-
D	D	D	DD	-
D	S	S	DSS	-

Syntax Example

The following DIVIDE instruction divides the value at location DATA by a value at a location defined by the label TAB plus the contents of index register 1. Both operands are single precision because no precision is specified.

```
DIVIDE   DATA,(TAB,#1)
```

DIVIDE - Divide integer values *(continued)*

Coding Example

The following example uses the DIVIDE instruction to determine the amount of time an experiment required in hours, minutes, and seconds. If the data area labeled TIME contained a value of 4796 (seconds), the first DIVIDE instruction would place a result of 1 in HOURS and would leave a remainder of 1196 in the first word of the TCB. The label of the TCB is TASK, the label of the PROGRAM statement.

The second DIVIDE instruction at label GETMINS would divide the remainder by 60 and place a result of 19 in MINUTES and a remainder of 56 in the TCB. This remainder represents the number of seconds and would be moved into SECONDS. The program would print out a final result of 1 hour, 19 minutes, and 6 seconds.

```
TASK      PROGRAM  START
START     EQU      *
            .
            .
NEXTIME   EQU      *
            .
            .
            .
GETHOURS  EQU      *
          DIVIDE   TIME,3600,RESULT=HOURS   NUMBER OF HOURS
GETMINS   EQU      *
          DIVIDE   TASK,60,RESULT=MINUTES   NUMBER OF MINUTES
GETSECS   EQU      *
          MOVE     SECONDS,TASK,(1,WORD)    GET REMAINDER
PRINTIME  EQU      *
          PRINTEXT ' ELAPSED TIME IN HOURS:MINUTES:SECONDS'
          PRINTNUM HOURS
          PRINTEXT ' : '
          PRINTNUM MINUTES
          PRINTEXT ' : '
          PRINTNUM SECONDS
          GOTO     NEXTIME                  CONVERT ANOTHER COUNT
            .
            .
            .
TIME      DATA     D'0'                     BEGINNING VALUE
HOURS     DATA     F'0'                     NUMBER OF ELAPSED HOURS
MINUTES   DATA     F'0'                     NUMBER OF ELAPSED MINUTES
SECONDS   DATA     F'0'                     NUMBER OF ELAPSED SECONDS
```

DO - Perform a program loop

The DO instruction begins a program loop. A loop is a set of one or more instructions that executes repeatedly until a condition you specify in the DO instruction is satisfied. You must end the DO loop with an ENDDO instruction.

You can code a loop within another loop. This technique is called "nesting." You can include up to 20 nested loops within your initial DO-ENDDO structure.

There are three forms of the DO instruction. DO UNTIL and DO WHILE provide a means of looping until or while a condition 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.

You also can use the DO instruction to perform a loop while or until a certain bit is 'on' (set to 1) or 'off' (set to 0).

The syntax box shows the DO UNTIL and DO WHILE forms of the DO instruction with a single conditional statement. You can specify several conditional statements, however, by using the AND and OR keywords. These keywords allow you to join conditional statements. The keywords are described in the operands list and examples using the keywords are shown under "Syntax Examples with DO and ENDDO" on page LR-130.

Syntax:

```
label      DO       count,TIMES,INDEX=,P1=
label      DO       UNTIL,(data1,condition,data2,width)
label      DO       WHILE,(data1,condition,data2,width)

Required:   count or one conditional statement
            with UNTIL or WHILE
Defaults:   width is WORD
Indexable:  count or data1 and data2 in each statement
```

Operand	***Description***
count	The number of times the loop is to be executed. You can specify a constant or the label of a variable. The maximum value is 32767. The system completes one loop each time it encounters the ENDDO instruction. **Note:** If count=0, the system executes the loop one time.
TIMES	This optional operand serves only as a comment for the count operand.
INDEX=	The label of a data area that the system resets to 0 before starting the DO loop and increases by 1 each time the instruction following the DO instruction executes. The first time the DO loop executes, the index has a value of 1.

DO - Perform a program loop *(continued)*

UNTIL This operand defines a loop that executes until the condition you specify is true. The loop executes at least once, even if the condition is initially true.

WHILE This operand defines a loop that executes as long as the condition you specify is true. The loop does not execute if the condition is initially false.

data1 The label of a data item to be compared to data2 or the label of the data area that contains the bit to be tested. This operand is valid only in a conditional statement with UNTIL or WHILE.

condition An operator that indicates the relationship or condition to be tested. Only code this operand in a conditional statement with UNTIL or WHILE. The valid operators for the DO instruction are as follows:

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

ON - Bit is 'on'
OFF - Bit is 'off'

data2 The data to be compared to data1 or the position, in data1, of the bit to be tested. Only code this operand in a conditional statement with UNTIL or WHILE. You can specify immediate data or the label of a variable. Immediate data can be an integer between 1 and 32768 or a hexadecimal value between 1 and 65535 (X'FFFF').

Bit 0 is the left-most bit of the data area.

width Specifies an integer number of bytes or one of the following:

BYTE - bytes
WORD - words (the default)
DWORD - doublewords
FLOAT - floating-points (one word, 2-byte value)
DFLOAT - doublewords floating-points (4-byte value)

Code this operand only in a conditional statement using UNTIL or WHILE. The default is WORD.

AND Enables you to join conditional statements when you code DO UNTIL or DO WHILE. Code the operand between the conditional statements you want to join. With DO UNTIL, the AND indicates that the loop should execute *until* all the conditional statements that the operand joins are true. With DO WHILE,

DO - Perform a program loop *(continued)*

the AND indicates that the loop should execute *while* all the conditional statements the operand joins are true.

You can join several pairs of conditional statements with several AND operands. You also can use the AND and OR operands within the same DO instruction.

OR Enables you to join conditional statements when you code DO UNTIL or DO WHILE. Code the operand between the conditional statements you want to join. With DO UNTIL, the OR indicates that the loop should execute *until* one of the conditional statements the operand joins is true. With DO WHILE, the OR indicates that the loop should execute *while* any of the conditional statements the operand joins is true. See the syntax examples for this instruction.

You can join several pairs of conditional statements with several OR operands. You also can use the AND and OR operands within the same DO instruction.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Rules for Evaluating Statement Strings Using AND and OR

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

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:

- The expression is evaluated from left to right.
- 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.
- If the condition is true and the next conjunction is **AND**, the next conjunction is checked.
- If the condition is false and the next conjunction is **OR**, the next condition is checked.
- If the condition is false and the next conjunction is **AND**, or if there are no more conjunctions, the string is false and the evaluation ceases.

DO - Perform a program loop *(continued)*

The order of the statements and the 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 the 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.

Syntax Examples with DO and ENDDO

See the IF instruction for more samples of conditional statements.

1) Perform a loop 100 times.

```
DO      100
 .
 .
 .
ENDDO
```

2) Perform a loop the number of times specified in N. The TIMES operand serves as a comment.

```
DO      N,TIMES
 .
 .
 .
ENDDO
```

3) Perform a loop until the first 4 bytes of A are less than the first 4 bytes of B.

```
DO       UNTIL,(A,LT,B,4)
 .
 .
 .
ENDDO
```

DO - Perform a program loop *(continued)*

4) Perform a loop until A contains a floating-point value equal to 1000.

```
DO      UNTIL,(A,EQ,1000,FLOAT)
 .
 .
 .
ENDDO
```

5) Perform a loop while the first word of B is not equal to the first word of C.

```
DO      WHILE,(B,NE,C)
 .
 .
 .
ENDDO
```

6) Perform a loop while the first 4 bytes of A are less than the first 4 bytes of B.

```
DO      WHILE,(A,LT,B,4)
 .
 .
 .
ENDDO
```

7) Perform a loop until the third bit starting at label A is a 1.

```
DO      UNTIL,(A,ON,2)
 .
 .
 .
ENDDO
```

8) Perform a loop until the bit number contained in BIT1, starting at label A, is a 0.

```
DO      UNTIL,(A,OFF,BIT1)
 .
 .
 .
ENDDO
```

9) Perform a loop until A equals B *and* A equals C.

```
DO      UNTIL,(A,EQ,B),AND,(A,EQ,C)
 .
 .
 .
ENDDO
```

DO - Perform a program loop *(continued)*

10) Perform a loop while A is not equal to 1, *or* while the first doubleword in D is equal to the first doubleword in E, *and* while register 1 is not equal to 14.

```
DO      WHILE,(A,NE,1),OR,(D,EQ,E,DWORD),AND,(#1,NE,14)
  .
  .
  .
ENDDO
```

11) This example shows a nested DO loop.

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

12) This example shows a nested DO loop that is also within an IF-ELSE-ENDIF structure.

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

DO - Perform a program loop *(continued)*

Coding Example

The following example shows three DO loops.

The first DO loop, at label D1, executes twice and ends. The second DO loop, at label D2, executes at least once and continues to loop until the value of INDEX1 is greater than or equal to 2.

The third DO loop, at label D3, executes as long as (WHILE) the value of INDEX2 is less than or equal to 1. If the condition is not initially true, the third loop does not execute at all.

```
         .
         .
         .
D1       DO     2,TIMES,INDEX=INDEX
           MOVE   INDEX1,0
D2         DO     UNTIL,(INDEX1,GE,2)
              ADD    INDEX1,1
              MOVE   INDEX2,0
D3            DO     WHILE,(INDEX2,LE,1)
                 ADD       INDEX2,1
                 PRINTNUM INDEX,3,3,4
              ENDDO
           ENDDO
         ENDDO
         .
         .
INDEX    DATA    F'1'
INDEX1   DATA    F'1'
INDEX2   DATA    F'1'
         .
         .
```

The above example generates the following printout:

```
1            1            1
1            1            2
1            2            1
1            2            2
2            1            1
2            1            2
2            2            1
2            2            2
```

DSCB - Create a data set control block

The DSCB statement creates a data set control block (DSCB). A DSCB provides the information the system requires to use a data set within a particular volume.

The first 3 words of every DSCB is an event control block (ECB). You can refer to fields within a DSCB by using the DSCB equate table, DSCBEQU.

Syntax:

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

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

Operand	***Description***
DS#=	The alphameric label which is used to refer to a DSCB in disk or tape I/O instructions. This label 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 you do not specify VOLSER. **Note:** If the DSCB is for a tape data set, you must specify VOLSER prior to DSOPEN. In addition, you must supply the 1 to 6 character tape drive ID if there is no volume label. The tape drive ID is assigned during system generation with the TAPE definition statement.
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 statement and then invoked with the CALL instruction. The $DISKUT3 is invoked with the LOAD instruction. For more information on DSOPEN and $DISKUT3 see Appendix D or refer to the *Event Driven Executive Language Programming Guide.*

DSCB - Create a data set control block *(continued)*

Syntax Example

The following DSCB statement creates a data set control block with the label INDATA.

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

ECB - Create an event control block

The ECB statement generates a 3-word event control block (ECB) that defines an event. The system places a value in the first word of the control block when an event has occurred. When the system signals the occurrence of an event in the ECB, the ECB is said to have been "posted."

Normally this statement is not needed for application programs you assemble with the host or Series/1 macro assemblers. The host and Series/1 macro assemblers automatically generate a control block for an event named in a POST instruction.

You must code the necessary ECBs in programs assembled under $EDXASM, except for those ECBs created when you code the EVENT= operand on the PROGRAM or TASK statement.

You can code a maximum of 25 ECB statements in a program. If your program requires more than 25 ECBs, you must create them using DATA statements. An example of how to create an ECB is shown following the description of this statement.

When coding the ECB statement, you can include a comment which will appear with the statement on your compiler listing. If you include a comment, you must also specify the code operand. The comment must be separated from the operand field by at least one blank and it may not contain commas.

Syntax:

```
label        ECB      code      comment

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

Operand	*Description*
label	The label of the event that you specify in a POST instruction.
code	Initial value of the code field (word 1). If this word is not a zero when a WAIT is issued, no wait occurs unless the WAIT has RESET coded.

ECB - Create an event control block *(continued)*

Syntax Example

The ECB statement:

```
ECB1        ECB
```

is equivalent to coding,

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

EJECT

EJECT - Continue compiler listing on a new page

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 you are using one.

You can place EJECT within executable instructions.

Syntax:

```
blank          EJECT

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

Operand	*Description*
none	none

Coding Example

See the PRINT statement for an example using EJECT.

ELSE - Specify action for a false condition

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 statement.

Syntax:

```
label          ELSE

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

Operand	*Description*
none	none

Syntax Examples

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

END

END - Signal end of source statements

The END statement signals the compiler that the program contains no further source statements.

END must be the last statement in a program, a separately compiled task, or a subroutine. Unpredictable results can occur if you do not code an END statement.

Syntax:

```
blank          END

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

Operand	*Description*
none	none

Coding Example

The following example enqueues $SYSLOG, prints the time and date, dequeues $SYSLOG, and ends. END is the last statement in the program.

```
PRINDATE    PROGRAM    START
START       EQU        *
            ENQT       $SYSLOG
            PRINTIME
            PRINDATE
            DEQT
            PROGSTOP
            ENDPROG
            END
```

ENDATTN - End attention-interrupt-handling routine

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

Syntax:

```
label          ENDATTN

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

Operand	***Description***
none	none

Coding Example

See the ATTNLIST statement for an example using the ENDATTN instruction.

ENDDO

ENDDO - End a program loop

The ENDDO statement defines the end of a DO loop. It must be preceded by a DO instruction.

Syntax:

```
label         ENDDO

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

Operand	*Description*
none	none

Coding Example

See the examples following the DO instruction.

ENDIF - End an IF-ELSE structure

The ENDIF statement 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
```

Operand	*Description*
none	none

Syntax Examples

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

ENDPROG - End a program

The ENDPROG statement ends a program. It must be the next to the last statement in your program (except when you include a $ID statement). The last statement must be END. You can code the RETURN= operand on the ENDPROG statement to acquire the system-return subroutine support without link-editing the subroutine with your program.

The ENDPROG statement generates a task control block (TCB) for the main program. You can locate the TCB by referring to the label on the PROGRAM statement.

Syntax:

```
blank         ENDPROG  RETURN=

Required:     none
Defaults:     RETURN=NO (if your program contains
              a USER instruction, the default is YES)
Indexable:    none
```

Operand	*Description*
RETURN=	RETURN=YES generates the $$RETURN subroutine in your program. $$RETURN enables you to return to an EDL program from an assembler subroutine when you code

```
BAL    RETURN,R1
```

in the assembler subroutine. When you specify RETURN=YES, it is not necessary to link-edit the $$RETURN subroutine to your program.

If your program has a USER instruction coded, then the RETURN operand is not necessary on the ENDPROG statement. The USER instruction causes the system module $$RETURN to be generated as part of your program.

RETURN=NO is the default value for the RETURN operand unless your program contains a USER instruction. If you code RETURN=NO or allow the default, the system module is not generated as part of your program.

RETURN=EXTRN generates an external reference to the system subroutine $$RETURN. If you code RETURN=EXTRN, you must link-edit the $$RETURN subroutine to your program.

ENDPROG - End a program *(continued)*

Syntax Example

The ENDPROG statement precedes the END statement.

```
          .
          .
          .
          PROGSTOP
FIELD     DATA        F'0'
MESSAGE   TEXT        'ENTER YOUR NAME :'
          ENDPROG
          END
```

ENDTASK - End a task

The ENDTASK instruction defines the end of a task. Each task, except the primary task, requires one ENDTASK as its final instruction. When this instruction executes, the task is detached. If another ATTACH is issued, execution begins at the first 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 system attaches the task.

Syntax:

```
label          ENDTASK  code,P1=

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

Operand	*Description*
code	The post code can be any 1-word value. This code will be inserted in the terminating ECB ($TCBEEC) of the task being detached. A complete list of TCB equates is in the *Internal Design*.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Coding Example

In the following example the main program, PROGA, attaches both TASKA and TASKB during execution. Both tasks must be coded within the main program; you cannot code the tasks in subprograms that are later link-edited with the main program. The main program code always ends with the ENDPROG and END statements (unless you code an intervening $ID statement). The task source code always ends with the ENDTASK statement.

The first ATTACH instruction starts TASKA. TASKA begins by setting its post code to -1. If an error occurs, the task ends with a post code of 999. The second ATTACH instruction starts TASKB.

The IF instruction at label CHECK examines the post code of TASKA to see if the task ended successfully. If the task did not end successfully, another ATTACH instruction reattaches TASKA. Because TASKA can only end with an ENDTASK statement, execution always resumes at the instruction following the BEGINA label.

If TASKB detaches at the DETACH instruction, execution resumes at the instruction following the DETACH. If TASKB detaches at the ENDTASK statement, the task resumes execution at BEGINB.

ENDTASK - End a task *(continued)*

```
PROGA      PROGRAM    START
START      EQU        *
             .
             .
           ATTACH     TASKA
             .
           ATTACH     TASKB
             .
CHECK      IF         ($TCBEEC+TASKA,NE,-1)
              ATTACH    TASKA
           ENDIF
             .
           ATTACH     TASKB
             .
           PROGSTOP
             .
TASKA      TASK       BEGINA
BEGINA     EQU        *
           MOVE       CODE,-1
             .
           IF         (RESULT,EQ,ERROR)
              MOVE      CODE,999
           ENDIF
           ENDTASK    1,P1=CODE
*
TASKB      TASK       BEGINB
BEGINB     EQU        *
           ADD        C,D
             .
           DETACH
             .
           ENDTASK
           ENDPROG
           END
```

ENQ - Gain exclusive control of a resource other than a terminal

The ENQ instruction gains exclusive control of a resource *other* than a terminal by acquiring control of the queue control block (QCB) associated with that resource. Use ENQ to gain control of logical or physical resources such as sensor-based I/O devices, subroutines, and data sets.

Note: Use the ENQT instruction to acquire exclusive use of any resource you define with a TERMINAL statement, such as a display station or printer.

When several programs need to use the same resource, the ENQ instruction can ensure serial (one at a time) use of the resource. Programs try to acquire control of, or "enqueue," a specific QCB before trying to use the resource. If the QCB is "busy," the program can wait for the resource to become available or execute another routine.

In general, there are two types of resources, system and user. System resources can be shared serially by all programs and are defined by labels that are known across the system. The QCBs associated with these resources must reside in $SYSCOM, the system common area. (Refer to the *Installation and System Generation Guide* for a discussion of $SYSCOM.) User resources are shared serially by different parts of one user program and are identified by labels known only within that program. The QCBs associated with these resources reside within the program.

You must define each QCB contained in a program compiled under $EDXASM with the QCB statement. The QCB statement generates the five-word queue control block in your program. The Series/1 and host macro assemblers automatically create a required QCB if you include a DEQ instruction naming the QCB in your program.

ENQ normally assumes that the QCB to be enqueued is in the same partition as the current program. However, your program can enqueue a QCB in another partition by using the cross-partition capability of ENQ. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example of enqueuing a resource in another partition. Refer to the *Event Driven Executive Language Programming Guide* for more information on cross-partition services.

Syntax:

```
label        ENQ         qcb,BUSY=,P1=

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

ENQ - Gain exclusive control of a resource other than a terminal *(continued)*

Operand	*Description*
qcb	The label of the QCB to be enqueued.
BUSY=	The label of the instruction to receive control if the QCB you try to enqueue is in use. If you do not code this operand and the QCB is in use, the system suspends the execution of your program until the resource associated with the QCB becomes available.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Coding Example

The following example shows the use of ENQ and DEQ instructions.

The ENQ instruction attempts to enqueue the queue control block labeled SBRTNQCB. If the first word of the QCB contains a zero, the subroutine labeled SUBRTN is being used by another program. The program, in this case, would wait for the resource to become available. If the first word of the QCB is not a zero, the program can call SUBRTN.

When SUBRTN ends, it places a code of 99 in RETURNCD. The DEQ instruction releases exclusive control of the QCB and places the value of RETURNCD (99) in the first word of the QCB. The nonzero value in the QCB serves as a signal to other programs that the resource associated with the QCB is available.

```
                  .
                  .
                ENQ       SBRTNQCB
                CALL      SUBRTN
                DEQ       SBRTNQCB,0,P2=RETURNCD
                  .
                  .
                SUBROUT   SUBRTN
                  .
                  .
                MOVE      RETURNCD,99
                RETURN
                  .
                  .
SBRTNQCB          QCB     -1
                  .
                  .
```

ENQT - Gain exclusive control of a terminal

The ENQT instruction acquires exclusive control of a terminal. To acquire exclusive control of a terminal is to "enqueue" it. A "terminal" is any device, such as a display station or printer, that you define with a TERMINAL statement during system generation.

Your program releases exclusive control of a terminal when it executes a DEQT or PROGSTOP instruction.

Once your program enqueues a terminal, it must release control of that terminal with a DEQT instruction before attempting to enqueue another terminal.

When coding the ENQT instruction, you can include a comment which will appear with the instruction on your compiler listing. If you include a comment, you must specify at least one operand with the instruction. The comment must be separated from the operand field by one or more blanks and it may not contain commas.

Syntax:

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

Required:    none
Defaults:    SPOOL=YES
             name - label of the terminal which is currently in use
                  by the program
Indexable:   none
```

Operand	***Description***
name	The label of an IOCB statement or one of two special device names: $SYSLOG or $SYSPRTR. $SYSLOG is the name of the system display station; $SYSPRTR is the name of the system printer. Your program enqueues the terminal from which you loaded it if you allow this operand to default.

When you specify $SYSLOG or $SYSPRTR, the system refers to the TERMINAL statement you set up for each of these devices during system generation. Do not code an IOCB statement for these devices.

When you want to specify a terminal other than $SYSLOG or $SYSPRTR, you can code the label of an IOCB statement for this operand. The ENQT instruction refers to the IOCB statement for the name of the terminal you want to control. The name on the IOCB statement is the name you assigned to the terminal during system generation. By referring to an IOCB statement, you also can redefine certain terminal characteristics. You can, for example, reset screen or page margins, or change a terminal from a roll screen device to a static screen device. (See the IOCB statement for a description of the terminal characteristics you can redefine.) The terminal characteristics you specify with an IOCB statement remain in effect until you release control of the terminal.

ENQT - Gain exclusive control of a terminal *(continued)*

BUSY= The label of the instruction to receive control if the terminal you try to enqueue is in use. If you do not code this operand and the terminal is in use, the system suspends the execution of your program until the terminal you request becomes available.

SPOOL= YES, the default, to allow the system to send spooled output to the spool device you enqueue when the spool facility is active. This operand has no effect if the spool facility is not active or if the device you enqueue is not a spool device.

NO, to prevent the system from sending spooled output to the spool device you enqueue when the spool facility is active.

This operand remains in effect until your program executes a DEQT or PROGSTOP instruction.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Special Considerations

You should note the following considerations when using the ENQT instruction:

- If your program has exclusive control of a terminal and loads another program, the system dequeues the terminal unless you coded DEQT=NO on the LOAD instruction. See "LOAD - Load a Program" on page LR-263for a description of the DEQT operand.
- ATTNLIST commands cannot gain access to an enqueued terminal.
- If your program attempts to enqueue a terminal it already controls, the ENQT instruction can change the characteristics of the terminal in use if it refers to an IOCB statement that defines new terminal characteristics.
- If an ENQT instruction refers to an IOCB that sets up the limits of a logical screen, the output for that screen starts at the top of the working area. The system, however, does not immediately move the cursor to this location. Your program can position the cursor at the top of the working area by issuing a TERMCTRL DISPLAY.
- To preserve the correct current line pointer when the system sends spooled output to an enqueued terminal, you must code a TERMCTRL DISPLAY as the last I/O instruction before your program issues an ENQT instruction that redefines the characteristics of that terminal.

ENQT - Gain exclusive control of a terminal *(continued)*

Syntax Examples

1) Enqueue the system printer, $SYSPRTR.

```
         ENQT    $SYSPRTR
          .
          .
         DEQT
```

2) Enqueue the device TTY1. The ENQT instruction refers to the IOCB labeled TERM1 for the name of the device. If TTY1 is not available, the program passes control to the label ALTERN and enqueues $SYSLOG.

```
TEST     PROGRAM  START
TERM1    IOCB     TTY1,PAGSIZE=24
START    EQU      *
         ENQT     TERM1,BUSY=ALTERN
          .
          .
         DEQT
          .
ALTERN   ENQT     $SYSLOG
          .
```

Coding Example

The first ENQT instruction in the program attempts to enqueue $SYSPRTR. If the device is busy, the program displays a message and attempts to enqueue an alternate printer ($SYSLIST). If the alternate printer is busy, the program waits for it. When the program obtains a printer, it executes the CALL instruction at the label GOTPRTR. The DEQT instruction at the label RELEASE releases exclusive control of the enqueued printer (either $SYSPRTR or $SYSLIST).

```
          .
          .
          .
GETPRTR   EQU       *
          ENQT      $SYSPRTR,BUSY=BUSYEXIT
          GOTO      GOTPRTR
BUSYEXIT  EQU       *
          PRINTEXT  '$SYSPRTR IS BUSY. ATTEMPTING TO ENQT ALTERNATE'
          ENQT      PRTRIOCB
GOTPRTR   EQU       *
          CALL      SUBRTN
           .
           .
RELEASE   EQU       *
          DEQT
          PROGSTOP
PRTRIOCB  IOCB      $SYSLIST
          ENDPROG
          END
```

ENTRY - Define a program entry point

The ENTRY statement defines one or more labels as being entry points within a program module. A maximum of 10 labels are allowed on one ENTRY statemant. These entry-point labels can be referred to by instructions in other program modules that are link-edited with the module that defines the entry-point label. The program modules that refer to an entry-point label must contain either an 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
```

Operand	*Description*
symbol	One or more symbols that appear as instruction labels within the program module.

ENTRY - Define a program entry point *(continued)*

Coding Example

In module A, the first ENTRY statement signifies that the program can be entered at label GETTIME. In module B, the entry defines label GOTTIME as being an entry point. Both of these labels are also used with EXTRN statements so that their addresses can be resolved when the two modules are link-edited together. The second ENTRY statement in module A will allow the time to be printed without the 'TIME IS NOW' text.

MODULE A

```
          .
          .
          ENTRY      GETTIME
          ENTRY      GETTIME2
          EXTRN      GOTTIME
          .
          .
GETTIME   EQU        *
          PRINTEXT   '@THE TIME IS NOW '
GETTIME2  EQU        *
          PRINTIME
          GOTO       GOTTIME
          .
          .
          .
```

MODULE B

```
          .
          .
          ENTRY      GOTTIME
          EXTRN      GETTIME
          .
          .
TIME      EQU        *
          GOTO       GETTIME
GOTTIME   EQU        *
          .
          .
          .
```

Note: The two ENTRY statements in module A could have been coded as follows:

```
          ENTRY   GETTIME,GETTIME2
```

EOR - Compare the binary values of two data strings

The Exclusive OR instruction (EOR) compares the binary value of operand 2 with the binary value of operand 1. The instruction compares each bit position in operand 2 with the corresponding bit position in operand 1 and yields a result, bit by bit, of 1 or 0. If the bits compared are the same, the result is 0. If the bits compared are not the same, the result is 1. If both input fields are identical, the resulting field is 0. If one or more bits differ, the resulting field contains a mixture of 0's and 1's.

Syntax:

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

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

Operand	*Description*
opnd1	The label of the data area to be compared with opnd2. Opnd1 cannot be a self-defining term. The system stores the result of the operation in this operand unless you code the RESULT operand.
	This operand can be a byte, word , or doubleword.
opnd2	The value compared with opnd1. You can specify a self-defining term or the label of a data area. This operand can be a byte, word, or doubleword.
count	The number of consecutive values in opnd1 on which the operation is to be performed. The maximum value allowed is 32767.

The count operand can include the precision of the data. Select one precision which the system uses for opnd1, opnd2, and the resulting bit string. When specifying a precision, code the count operand in the form,

```
(n,precision)
```

where "n" is the count and "precision" is one of the following:

BYTE -- byte precision
WORD -- word precision (default)
DWORD -- doubleword precision

The precision you specify for the count operand is the portion of opnd2 that is used in the operation. If the count is (3,BYTE), the system compares the first byte of data in opnd2 to the first three bytes of data in opnd1.

EOR - Compare the binary values of two data strings *(continued)*

RESULT= The label of a data area or vector in which the result is to be placed. When you specify RESULT, the value of opnd1 does not change during the operation. This operand is optional.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) The EOR instruction compares the first byte of data in D to the first byte of data in C and places the result in R.

```
          EOR    C,D,(1,BYTE),RESULT=R
           .
           .
           .
C         DATA   X'92'         binary 1001 0010
D         DATA   X'8F'         binary 1000 1111
R         DATA   X'00'
```

After the operation, R contains:

Hexadecimal -- X'1D'

Binary -- 0001 1101

2) The EOR instruction compares the first byte of data in OPER2 to the first three bytes of data in OPER1. The result of the operation is stored in RESULTX.

```
          EOR      OPER1,OPER2,(3,BYTE),RESULT=RESULTX
           .
           .
           .
OPER1     DC       X'00'       binary 0000 0000
          DC       X'A5'       binary 1010 0101
          DC       X'01'       binary 0000 0001
OPER2     DC       X'FF'       binary 1111 1111
RESULTX   DC       2F'0'
           .
           .
```

After the operation, RESULTX contains:

Hexadecimal -- X'FF5A FE00'

Binary -- 1111 1111 0101 1010 1111 1110 0000 0000

EOR - Compare the binary values of two data strings *(continued)*

3) The EOR instruction compares the first byte of data in TEST to the first three bytes of data in INPUT. The result of the operation is stored in OUTPUT.

```
         EOR     INPUT,TEST,(3,BYTE),RESULT=OUTPUT
          .
          .
          .
INPUT    DC      C'1.2'      binary  1111 0001 0100 1010 1111 0010
TEST     DC      C'0.0'      binary  1111 0000
OUTPUT   DC      3C'0'       binary  1111 0000 1111 0000 1111 0000
          .
          .
```

After the operation, OUTPUT contains:

Binary -- 0000 0001 1011 1010 0000 0010

EQU - Assign a value to a label

The EQU statement assigns a value to a label. The value is a word in length. You can use the label you define with the EQU statement as an operand in other instructions that permit the use of labels. The 'value' the statement assigns, or equates, to a label can consist of an integer constant, another label, an expression containing an arithmetic operator (for example, A+2), or an asterisk (*). See "Syntax Rules" on page LR-7 for a description of the four arithmetic operators: + (plus), - (minus), * (multiply), and / (divide).

Syntax:

```
label          EQU      value

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

Operand	*Description*
label	The label to be assigned a value. Do not define this label elsewhere in your program.
value	An integer constant, another label, an expression containing an arithmetic operator, or an asterisk (*). The asterisk points to the next available storage location in a program. It allows you to generate convenient labels that you can use within your program. Do not confuse this use of an asterisk with the arithmetic operator that signifies multiplication (*).

Your program must define any labels you code for this operand before the system processes the EQU statement. For example, if you code:

```
A      EQU      B
```

you must have previously defined the label B in your program.

Special Considerations

Here are some things to consider when you use the EQU statement in your program:

- When you use the label on the EQU statement as an operand in another instruction, the system interprets the label as a storage address unless you include a plus (+) sign before it. The system interprets a label preceded by a plus sign as a constant.
- Because EQU assigns a word value to a label, a byte-precision move of a label preceded by a plus sign would only move the leftmost byte of the word. If you equated the label A to the value 4 (X'0004'), for example, the system would move only the value X'00'.

EQU - Assign a value to a label *(continued)*

- If you equate a DATA or DC statement with a label, the system interprets the label as the address of the DATA or DC statement. If you try to use this label with a plus sign, however, the label will no longer point to the data when the load point of the program changes.

- You can equate a hexadecimal value to a label if the value can fit in a word (for example, X'FED1'). You can also equate one or two EBCDIC characters with a label (for example, C'AB'). You cannot form EQU expressions with the following types of data: H, D, E, and A. (See DATA/DC for a description of each of these data types.)

Syntax Examples

1) Assign a value of 2 (X'0002') to A.

```
A          EQU      2
```

2) Assign the value of A to label B. If A has a value of 5 (X'0005'), B also has a value of 5.

```
B          EQU      A
```

3) Assign the value of B plus 2 bytes to label A.

```
A          EQU      B+2
```

4) CALLA is equivalent to CALLSUB. The asterisk (*) points to the next available storage location in the program.

```
           GOTO     CALLA
             .
             .
             .
CALLA      EQU      *
CALLSUB    CALL     PROGA
```

5) Move the contents at address X'0002' to C.

```
A          EQU      2
           MOVE     C,A
```

6) Move A, a value of 2, to C.

```
A          EQU      2
           MOVE     C,+A
```

EQU - Assign a value to a label *(continued)*

7) Move 7 to the indexed location of A plus #1.

```
A         EQU      2
          MOVE     (A,#1),7
```

8) Add the value of C (X'0002') to D (X'0008'). The example defines the labels B and A before they appear in the EQU statements.

```
SAMPLE  PROGRAM   START
B       DATA      F'2'
START   EQU       *
        .
        .
        .
C       EQU       B
        ADD       D,C
        PROGSTOP
A       DATA      F'8'
D       EQU       A
```

9) A has a word value of X'0005'. The leftmost byte (value X'00') moves to location C.

```
A       EQU     5
        MOVE    C,+A,(1,BYTE)
```

10) Equate C to the address of F'0'. Move a value of 0 into TEMP.

```
C       EQU     *
        DATA    F'0'
        MOVE    TEMP,C
```

11) HERE has a value of 20. Move a value of 0 to address X'0014'.

```
HERE    EQU     20
        MOVE    HERE,0
```

EQU - Assign a value to a label *(continued)*

Coding Example

The following program moves data from three storage locations labeled A, C, and E. Label A is equal to the address of B times 2. Label C is equal to the address of D divided by 4. Label E is equal to the address of F divided by 5.

If the address of B is X'0052', the arithmetic expression B*2 refers to address X'00A4'. If the address of D is X'0054', the arithmetic expression D/4 refers to address X'0015'. For label F, if the address is X'0056', the arithmetic expression F/5 yields the address X'0017'. The system disregards the remainder in an arithmetic expression using the divide operator.

```
OPERATOR    PROGRAM     START
START       EQU         *
             .
             .
             .
M1          MOVE        HOLD1,A
M2          MOVE        HOLD2,C
M3          MOVE        HOLD3,E
             .
             .
             .
            PROGSTOP
HOLD1       DATA        F'0'
HOLD2       DATA        F'0'
HOLD3       DATA        F'0'
B           DATA        F'1'
D           DATA        F'2
F           DATA        F'3'
******************************
A           EQU         B*2
C           EQU         D/4
E           EQU         F/5
******************************
            ENDPROG
            END
```

ERASE - Erase portions of a display screen

The ERASE instruction clears or blanks a portion of a display screen. The instruction is only for terminals that have static screens. You can specify a static screen with the SCREEN operand of the TERMINAL statement or the IOCB instruction.

With a 4978, 4979 or 4980 terminal, the ERASE instruction clears a portion of the screen by setting that portion to a no data (null characters) condition. For a 3101 terminal in block mode, the instruction normally clears a portion of the screen by writing unprotected blanks to that area.

The ERASE instruction works differently on a 4978, 4979, or 4980 terminal than it does on a 3101 terminal in block mode. These differences are described under "3101 Display Considerations" on page LR-164.

The supervisor places a return code in the first word of the task control block (taskname) whenever an ERASE instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

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
```

Operand	***Description***
count	The number of bytes to be erased. Both nonprotected and protected characters contribute to the count, even if only nonprotected characters are to be erased. The ERASE instruction can erase up to an entire logical screen.
MODE=	FIELD, to end the erase operation when the display characters change from nonprotected to protected, or when the operation reaches the end of the current line.
	LINE, to end the erase operation at the end of the current line.
	SCREEN, to end the erase operation at the end of the logical screen.
	When the ERASE instruction erases the number of bytes you specified for the count, the operation will end even though the condition you specified on the MODE operand is not satisfied. The MODE operand determines the end of the

ERASE - Erase portions of a display screen *(continued)*

erase operation if you do not code a count value or if the condition you specify for MODE= occurs before the instruction erases the number of bytes in count.

TYPE= DATA, to erase only unprotected characters.

ALL, to erase both protected and unprotected characters.

SKIP= The number of lines to be skipped before the system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen.

For printers, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your static screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACE=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

ERASE - Erase portions of a display screen *(continued)*

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

3101 Display Considerations

The following considerations apply to the use of the ERASE instruction on a 3101 terminal in block mode.

If you code an ERASE instruction in with TYPE=DATA, the system ignores the count value. The instruction erases from the current cursor position to the end of the screen, clearing all unprotected data.

If you code TYPE=ALL on the ERASE instruction, the erase operation ends when the instruction erases the number of bytes in count, or when the operation reaches the end of a logical screen (whichever happens first). The default for count, when you code TYPE=ALL, is from the current cursor position to the end of the screen.

The system clears the entire 3101 screen if the cursor is in the home position (line zero,space zero), and an ERASE instruction with a count of 1920 executes.

The MODE operand on the ERASE instruction is affected by the TYPE operand in the following ways:

- MODE defaults to MODE=SCREEN if you code TYPE=DATA. The system forces the MODE operand to SCREEN even if you code MODE=LINE or MODE=FIELD.
- You can code the MODE=SCREEN or MODE=LINE if you code TYPE=ALL.
- The system forces the MODE operand to MODE=LINE if you code MODE=FIELD with TYPE=ALL.

If you code an ERASE instruction after a READTEXT instruction and the READTEXT buffer or TEXT statement is smaller than the number of characters actually transmitted by the 3101, you will need a delay between the READTEXT and ERASE instructions. The delay is necessary because your program should not issue an ERASE instruction until the 3101 completes sending the screen buffer. Depending on your application, you can use either an STIMER or WAIT KEY instruction to cause the delay.

ERASE - Erase portions of a display screen *(continued)*

Syntax Examples

1) Erase 4 bytes of unprotected data. End operation if protected data or the end of the line is reached.

```
ERASE   4,MODE=FIELD,TYPE=DATA
```

2) Erase the entire screen of protected and unprotected data.

```
ERASE   LINE=0,SPACES=0,MODE=SCREEN,TYPE=ALL
```

3) Erase all protected and unprotected data on line 1 of the screen.

```
ERASE   LINE=1,MODE=LINE,TYPE=ALL
```

Coding Examples

1) The following example is part of a program a company uses to update its personnel files. The example shows how you can use the ERASE instruction to erase portions of a display screen.

The example begins by enqueuing the terminal from which the program is loaded. The ENQT instruction refers to the label of an IOCB instruction that sets up a static screen for the terminal. This example assumes that the enqueued terminal is a 4978 or 4980.

The ERASE instruction at label E1 clears the entire screen, erasing both protected and unprotected characters (TYPE=ALL). Once the program erases the screen, it asks the operator to enter the employee's name and address in the three fields it displays on the screen. The WAIT key at label W1 prevents the program from reading the data until the operator presses the enter key. When the operator presses the enter key, the first READTEXT instruction reads in the data from the name field, the second READTEXT instruction reads in the data from the street field, and the third READTEXT instruction reads in data from the city field.

After the READTEXT instructions execute, the ERASE instructions at labels E2 through E4 erase all the data the operator entered on the screen. The ERASE instruction at label E2 clears the name field and ends after erasing 71 bytes of unprotected data. The count value overrides the MODE=SCREEN operand. The ERASE instruction at label E3 defaults to MODE=FIELD and clears the street field. The instruction stops erasing when it reaches the end of the line. The last ERASE instruction at label E4 clears the city field and continues to erase to the end of the line because MODE=LINE is coded.

ERASE - Erase portions of a display screen *(continued)*

```
                 .
                 .
                 .
         ENQT     TERMINAL
E1       ERASE    MODE=SCREEN,TYPE=ALL,LINE=0
         PRINTEXT MSG1,LINE=4,SPACES=2,PROTECT=YES
         PRINTEXT MSG2,LINE=5,SPACES=2,PROTECT=YES
         PRINTEXT FIELD1,LINE=6,SPACES=2,PROTECT=YES
         PRINTEXT FIELD2,LINE=7,SPACES=2,PROTECT=YES
         PRINTEXT FIELD3,LINE=8,SPACES=2,PROTECT=YES
W1       WAIT     KEY
         READTEXT NAME,LINE=6,SPACES=11,MODE=LINE
         READTEXT STREET,LINE=7,SPACES=11,MODE=LINE
         READTEXT CITY,LINE=8,SPACES=11,MODE=LINE
E2       ERASE    71,MODE=SCREEN,TYPE=DATA,LINE=6,SPACES=11
E3       ERASE    LINE=7,SPACES=11
E4       ERASE    MODE=LINE,LINE=8,SPACES=11
         DEQT
         PROGSTOP
TERMINAL IOCB     SCREEN=STATIC
MSG1     TEXT     'ENTER EMPLOYEE'S NAME, STREET ADDRESS, AND CITY'
MSG2     TEXT     'IN THE LABELED FIELDS.  PRESS ENTER WHEN FINISHED'
FIELD1   TEXT     ' NAME   : '
FIELD2   TEXT     ' STREET: '
FIELD3   TEXT     ' CITY   : '
NAME     TEXT     LENGTH=40
STREET   TEXT     LENGTH=60
CITY     TEXT     LENGTH=30
         ENDPROG
         END
```

2) The example that follows is similar to Example 1 but uses a 3101 terminal in block mode. The example begins by enqueuing the 3101 terminal. The IOCB instruction labeled TERMINAL sets up a static screen and a temporary I/O buffer for the device. The buffer area, labeled BUFFER, is 1920 bytes long.

As shown in Example 1, the ERASE instruction at label E1 erases the entire screen of protected and unprotected data. The program then issues a message asking the operator to enter the employee's name and address in three fields: NAME, STREET, and CITY. The program creates unprotected fields for the operator's input with the PRINTEXT instructions at labels P1, P2, and P3.

The WAIT key at label W1 prevents the program from reading the data until the operator presses the SEND key. When the operator presses the SEND key, the READTEXT instruction reads the entire display screen (protected and unprotected data) into the buffer area. A READTEXT instruction on 3101 in block mode starts reading at the beginning of the display screen if it does not issue a prompt message. The program reads the entire screen into the buffer area and then moves the desired data from the name, street, and city fields into three text buffers.

The ERASE instructions at label E2 through E4 erase all the employee data the operator entered on the screen. TYPE=ALL is coded on the ERASE instructions so that the count operand is not ignored. The ERASE instruction at label E2 clears the name field and ends after

ERASE - Erase portions of a display screen *(continued)*

erasing 71 bytes of unprotected and protected data. The count value overrides the MODE=SCREEN operand. The ERASE instruction at label E3 clears the street field and also ends after erasing 71 bytes of protected and unprotected data. Because the instruction has TYPE=ALL, the system changes the default MODE=FIELD to MODE=LINE. The last ERASE instruction at label E4 clears the city field and ends after erasing 20 bytes of protected and unprotected data.

Note: The coding of the data fields in this example differs slightly from Example 1 to allow for the attribute byte at the beginning of each field.

```
         .
         .
         .
         ENQT      TERMINAL
E1       ERASE     MODE=SCREEN,TYPE=ALL,LINE=0
         PRINTEXT  MSG1,LINE=4,SPACES=1,PROTECT=YES
         PRINTEXT  MSG2,LINE=5,SPACES=1,PROTECT=YES
         PRINTEXT  FIELD1,LINE=6,SPACES=2,PROTECT=YES
P1       PRINTEXT  NAME,LINE=6,SPACES=10,PROTECT=NO
         PRINTEXT  FIELD2,LINE=7,SPACES=2,PROTECT=YES
P2       PRINTEXT  STREET,LINE=7,SPACES=10,PROTECT=NO
         PRINTEXT  FIELD3,LINE=8,SPACES=2,PROTECT=YES
P3       PRINTEXT  CITY,LINE=8,SPACES=10,PROTECT=NO
W1       WAIT      KEY
         READTEXT  BUFFER,TYPE=ALL,MODE=LINE,LINE=0,SPACES=0
         MOVEA     #1,BUFFER
         MOVE      NAME,(492,#1),(40,BYTES)
         MOVE      STREET,(572,#1),(60,BYTES)
         MOVE      CITY,(652,#1),(7,BYTES)
E2       ERASE     71,MODE=SCREEN,TYPE=ALL,LINE=6,SPACES=11
E3       ERASE     71,LINE=7,SPACES=11,TYPE=ALL
E4       ERASE     20,MODE=SCREEN,LINE=8,SPACES=11,TYPE=ALL
         DEQT
         PROGSTOP
TERMINAL IOCB      SCREEN=STATIC,BUFFER=BUFFER
MSG1     TEXT      'ENTER EMPLOYEE'S NAME, STREET ADDRESS, AND CITY'
MSG2     TEXT      'IN THE LABELED FIELDS.  PRESS ENTER WHEN FINISHED'
FIELD1   TEXT      'NAME  :'
FIELD2   TEXT      'STREET:'
FIELD3   TEXT      'CITY  :'
NAME     TEXT      LENGTH=40
STREET   TEXT      LENGTH=60
CITY     TEXT      LENGTH=30
BUFFER   BUFFER    1920,BYTES
         ENDPROG
         END
```

EXCLOSE - Close an EXIO device

The EXCLOSE instruction closes, or disables, an EXIO device that you opened with the EXOPEN instruction.

Syntax:

```
label          EXCLOSE devaddr,ERROR=,P1=,P2=

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

Operand	***Description***
devaddr	The device address. Specify two hexadecimal digits.
ERROR=	The label of the first instruction to be executed if an error occurs during the execution of this instruction.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Close the EXIO device at the address X'08'.

```
EXOPEN   08,EXIOADDR
EXIO     PREPARE
  .
  .
  .
EXCLOSE  08
```

EXIO - Execute I/O

The EXIO instruction executes a command in an immediate device control block (IDCB) that you define using the IDCB statement.

Syntax:

```
label         EXIO      idcb,ERROR=,P1=

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

Operand	***Description***
idcb	The label of an IDCB statement.
ERROR=	The label of the first instruction to be executed if an error occurs during the operation. 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). **Note:** If the ECB being posted has not been reset, then the system posts the ECB provided for posting after an exception interrupt. 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 which points to an IDCB instruction with COMMAND=RESET.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Coding Example

In the following example, the first instruction (EXOPEN) specifies that, for the device at address X'08', information returned after an EXIO device interrupt is to be returned at the addresses pointed to by the 3 words following the EXIOADDR label.

The first EXIO instruction prepares the device at address X'08' so that it may interrupt on level 1.

The second EXIO instruction resets the device so that any incomplete I/O operation is ended.

The third EXIO instruction issues a START I/O command with the IDCB labeled STARTRD. The STARTRD IDCB uses the DCB labeled WRITEDCB. WRITEDCB is built for an ACCA

EXIO - Execute I/O *(continued)*

device so that a WRITE operation will be executed with the receiving station having the capability to BREAK the transmission. The TIMER1 (PRE and POSTTRANSMIT DELAYS) value is set to 33 milliseconds and the TIMER2 value (HALF-DUPLEX TURNAROUND) is set to 6.6 milliseconds. There is to be no DCB chaining and 12 bytes of data are to be transmitted starting at the address labeled MSG.

```
OPEN      EQU      *
          EXOPEN   08,EXIOADDR
          EXIO     PREPARE
            .
            .
          EXIO     RESET
            .
            .
          EXIO     STARTRD,ERROR=IOERROR
          EXCLOSE  08
            .
            .
IOERROR   EQU      *
          PRINTEXT '@IOERROR OCCURRED DURING INITIALIZATION@'
            .
            .
MSG       DATA  X'54484953'
          DATA  X'20414E20'
          DATA  X'41534349'
*
PREPARE   IDCB  COMMAND=PREPARE,ADDRESS=08,LEVEL=1,IBIT=1
RESET     IDCB  COMMAND=RESET,ADDRESS=08
STARTRD   IDCB  COMMAND=START,ADDRESS=08,DCB=WRITEDCB
*
WRITEDCB  DCB   IOTYPE=OUTPUT,DEVMOD=03,DVPARM1=0,DVPARM2=0002,       X
               DVPARM3=000A,DVPARM4=0, CHAINAD=0,COUNT=12,DATADDR=MSG
*
EXIOADDR  DATA  A(EXIO1)            POINTER TO 3 WORD
*                                   INTERRUPT BLOCK
          DATA  A(EXECBS)           ADDRESS OF ECB ADDRESSES
          DATA  A(EXSCSDCB)         ADDRESS OF START CYCLE
*                                   STEAL STATUS DCB
EXIO1     DATA  F'0'                INTERRUPT ID WORD
          DATA  F'0'                LSR AT INTERRUPT
          DATA  F'0'                ADDRESS OF ECB POSTED
*
EXECBS    DATA  A(EXCEND)           CONDITION CODE 0 ECB ADDR
          DATA  F'0'                NOT USED
          DATA  A(EXEXECP)          CONDITION CODE 2 ECB
          DATA  A(EXDEND)           CONDITION CODE 3 ECB ADDR
*
EXSCSDCB  DCB   IOTYPE=INPUT,COUNT=6,DATADDR=EXSCSWDS
*                                   START CYCLE STEAL STATUS DCB
EXSCSWDS  DATA  3F'0'
EXCEND    ECB   0                   CONTROLLER END ECB
EXEXECP   ECB   0                   EXCEPTION ECB
EXDEND    ECB   0                   DEVICE END ECB
```

Note: Additional examples using EXIO are shown in the *Customization Guide*.

EXIO - Execute I/O *(continued)*

Return Codes

The following codes are issued by the EXIO and EXOPEN instructions, and are returned in word 0 of the TCB. Word 1 of the TCB contains the supervisor instruction address.

Return Code	Condition
-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.
17	Device not opened or already closed.
18	Attempt to read or write to dynamic partition rejected. Use a static partition.

EXIO - Execute I/O *(continued)*

Interrupt Codes

The following codes are issued when an EXIO instruction was completed successfully, but the hardware performing the operation encountered an error. The hardware interrupt condition codes are returned in bits 4 - 7 of the ECB (word 0). If bit 0 is on, then bits 8 - 15 equal the device address.

Return Code	Condition
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).

EXOPEN - Open an EXIO device

The EXOPEN instruction opens an EXIO device and specifies 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
```

Operand	*Description*
devaddr	The device address. Specify two hexadecimal digits.
listaddr	The label of a 3-word list containing the following addresses:

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

1. Interrupt ID word
2. Level status register at time of the interrupt
3. Address of ECB posted.

Note: If this address is zero, the information is not returned.

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

Bit 0 of the posting code is on (1). Bits 4 to 7 contain the ccode; bits 8 to 15 contain the device address.

Interrupt condition codes are shown in "Return Codes" on page LR-171.

Word 3 The address of a DCB statement 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

EXOPEN - Open an EXIO device *(continued)*

from this device. If this address is zero, the operation is not 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 "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

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.

Coding Example

The EXOPEN instruction specifies that, for the device at address X'08', information returned after an EXIO device interrupt is to be returned at the addresses pointed to by the 3 words following the EXIOADDR label.

```
OPEN      EQU      *
          EXOPEN   08,EXIOADDR
            .
            .
            .
          EXCLOSE  08
            .
            .
EXIOADDR  DATA  A(EXIO1)           POINTER TO 3 WORD
*                                  INTERRUPT BLOCK
          DATA  A(EXECBS)          ADDRESS OF ECB ADDRESSES
          DATA  A(EXSCSDCB)        ADDRESS OF START CYCLE
*                                  STEAL STATUS DCB
EXIO1     DATA  F'0'               INTERRUPT ID WORD
          DATA  F'0'               LSR AT INTERRUPT
          DATA  F'0'               ADDRESS OF ECB POSTED
*
EXECBS    DATA  A(EXCEND)          CONDITION CODE 0 ECB ADDR
          DATA  F'0'               NOT USED
          DATA  A(EXEXECP)         CONDITION CODE 2 ECB
          DATA  A(EXDEND)          CONDITION CODE 3 ECB ADDR
*
EXSCSDCB  DCB   IOTYPE=INPUT,COUNT=6    START CYCLE STEAL
*                                       STATUS DCB
EXSCSWDS  DATA  3F'0'
EXCEND    ECB   0                  CONTROLLER END ECB
EXEXECP   ECB   0                  EXCEPTION ECB
EXDEND    ECB   0                  DEVICE END ECB
```

Return Codes and Interrupt Codes

For a list of return codes and interrupt condition codes, see the EXIO instruction.

EXTRN - Resolve external reference symbols

The EXTRN and WXTRN statements identify labels that are not defined within an object module. These labels reside in other object modules that will be link-edited to the module containing the EXTRN or WXTRN statements. The system resolves the reference to an EXTRN or WXTRN label when you link-edit the object module containing the EXTRN or WXTRN statement with the module that defines the label. The module that defines the label must contain an ENTRY statement for that label. (See the ENTRY statement for more information.)

If the system cannot resolve a label during the link-edit, it assigns the label the same address as the beginning of the program. You can include up to 255 EXTRN and WXTRN symbols in your program.

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

Only labels defined by EXTRN statements are used as search arguments during the AUTOCALL processing function of $EDXLINK. Any additional external labels found in the module found by AUTOCALL are used to resolve both EXTRN and WXTRN labels. Refer to the description of $EDXLINK in the *Operator Commands and Utilities Reference* for further information.

The main difference between the WXTRN and EXTRN statements is that you must resolve an EXTRN label at link-edit time. It is not necessary to resolve a WXTRN label at link-edit time. The unresolved label coded as an EXTRN receives an error return code from the link process. The same unresolved label coded as a WXTRN receives a warning return code. Both the error and the warning codes indicate unresolved labels. If you know that your application program does not need a label resolved, code it as a WXTRN and your program should execute successfully. Your application will not execute correctly, however, if you try to reference an unresolved label coded in your application program as a WXTRN.

Syntax:

```
blank        EXTRN    label
blank        WXTRN    label
Required:    one label
Defaults:    none
Indexable:   none
```

Operand	***Description***
label	An external label. You can code up to 10 labels, separated by commas, on a single EXTRN or WXTRN statement.

EXTRN - Resolve external reference symbols *(continued)*

Coding Example

The following coding example shows a use of the EXTRN statement.

The labels DATA1, DATA2, LABEL1, and LABEL2 are defined outside this module. The ADD instruction adds the values at DATA1 and DATA2 although the values are defined outside the module where they are being added. The GOTO instructions also can pass control to the the two externally defined labels, LABEL1 and LABEL2.

Each of the external labels could have been entered on a separate line or all three of the EXTRN labels could have been entered with a single EXTRN statement.

```
          .
          .
          .
          EXTRN    DATA1,DATA2
          EXTRN    LABEL1
          WXTRN    LABEL2
          .
          .
          ADD      DATA1,DATA2,RESULT=INDEX
          IF       (INDEX,GT,6)
             GOTO  LABEL1
          ELSE
             GOTO  LABEL2
          ENDIF
          .
          .
INDEX     DATA     F'0'
          .
          .
```

FADD - Add floating-point values

The floating-point add instruction (FADD) adds a floating-point value in operand 2 to a floating-point value in operand 1. You can use positive or negative values.

You must code FLOAT=YES on the PROGRAM statement of a program using floating-point instructions in its initial task and on the TASK statement of every 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
```

Operand	*Description*
opnd1	The label of the data area to which opnd2 is added. Opnd1 cannot be a self-defining term. The system stores the result of the operation in opnd1 unless you code the RESULT operand.
opnd2	The value added to opnd1. You can specify a self-defining term or the label of a data area. The valid range for this operand is from -32768 to +32767.
RESULT=	The label of a data area in which the result is to be placed. When you specify RESULT, the value of opnd1 does not change during the operation. This operand is optional.
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 the RESULT operand was coded. Permissible characters are:

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

The default is single precision.

FADD - Add floating-point values *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Index Registers

You cannot use the index registers (#1 and #2) as operands in floating-point operations because they are only 16 bits in length. You can, however, use the software registers to specify the address of a floating-point operand.

Syntax Examples

1) The FADD instruction adds two single-precision floating-point values and stores the result in RESULTF.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FADD      OP1F,OP2F,RESULT=RESULTF,PREC=FFF
            .
            .
            .
OP1F      DC        E'1.5'
OP2F      DC        E'0.2'
RESULTF   DC        E'0'
```

After the FADD operation, RESULTF contains the value 1.70 .

2) The FADD instruction adds two extended-precision floating-point values and stores the result in RESULTL.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FADD      OP1L,OP2L,RESULT=RESULTL,PREC=LLL
            .
            .
            .
OP1L      DC        L'50000.5'
OP2L      DC        L'40.4'
RESULTL   DC        L'0'
```

After the FADD operation, RESULTL contains the value 50040.90 .

FADD - Add floating-point values *(continued)*

3) The FADD instruction adds two single-precision floating-point values written in exponent (E) notation. The result is stored in RESULTFE.

```
FLOAT     PROGRAM    START,FLOAT=YES
            .
            .
          FADD       OP1FE,OP2FE,RESULT=RESULTFE,PREC=FFF
            .
            .
            .
OP1FE     DC         E'2.5E+1'       Equals decimal 25.0
OP2FE     DC         E'0.5E-1'       Equals decimal .05
RESULTFE  DC         E'0'
```

After the FADD operation, RESULTFE contains the value .2505E+02 . This value is equal to the decimal value 25.05 .

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). You must test for the return code immediately after the floating-point instruction is executed or the code may be destroyed by following instructions.

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

FDIVD - Divide floating-point values

The floating-point divide instruction (FDIVD) divides a floating-point value in operand 1 by a floating-point value in operand 2. You can use positive or negative values.

You must code FLOAT=YES on the PROGRAM statement of a program that uses floating-point instructions in its initial task and on the TASK statement of every 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
```

Operand	*Description*
opnd1	The label of the data area containing the value divided by opnd2. Opnd1 cannot be a self-defining term. The system stores the result of the operation in opnd1 unless you code the RESULT operand.
opnd2	The value by which opnd1 is divided. You can specify a self-defining term or the label of a data area. The valid range for this operand is from -32768 to +32767.
RESULT=	The label of a data area in which the result is to be placed. When you code RESULT, the value of opnd1 does not change during the operation.
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 the RESULT operand was coded. Permissible characters are:

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

The default is single precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

FDIVD - Divide floating-point values *(continued)*

Index Registers

You cannot use the index registers (#1 and #2) as operands in floating-point operations because they are only 16 bits in length. You can, however, use the software registers to specify the address of a floating-point operand.

Syntax Examples

1) The FDIVD instruction divides two single-precision floating-point values and stores the result in RESULTF.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FDIVD     OP1F,OP2F,RESULT=RESULTF,PREC=FFF
            .
            .
            .
OP1F      DC        E'1.5'
OP2F      DC        E'0.2'
RESULTF   DC        E'0'
```

After the FDIVD operation, RESULTF contains the value 7.50 .

2) The FDIVD instruction divides two extended-precision floating-point values and stores the result in RESULTL.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FDIVD     OP1L,OP2L,RESULT=RESULTL,PREC=LLL
            .
            .
            .
OP1L      DC        L'50000.5'
OP2L      DC        L'40.4'
RESULTL   DC        L'0'
```

After the FDIVD operation, RESULTL contains the value 1237.64 .

FDIVD - Divide floating-point values *(continued)*

3) The FDIVD instruction divides two single-precision floating-point values written in exponent (E) notation. The result is stored in RESULTFE.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FDIVD     OP1FE,OP2FE,RESULT=RESULTFE,PREC=FFF
            .
            .
            .
OP1FE     DC        E'2.5E+1'       Equals decimal 25.0
OP2FE     DC        E'0.5E-1'       Equals decimal .05
RESULTFE DC         E'0'
```

After the FDIVD operation, RESULTFE contains the value .5000E+03 . This value is equal to the decimal value 500 .

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). You must test for the return code immediately after the floating-point instruction is executed or the code may be destroyed by following instructions.

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

FIND - Locate a character

The FIND instruction searches a character string for the first occurrence of a specific character (byte).

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
```

Operand	*Description*
character	The character that is the object (target) of the search. You can specify a text character or a hexadecimal value.
string	The label of the string to be searched. The search will begin at the address of the label.
length	The number of bytes to be searched. You can code a positive integer or the label of a data area containing a positive integer.
where	The label of a data area where the address of the target character is to be stored if it is found. If the target character is not found, this data area remains unchanged.
notfound	The label of the instruction to be executed if the target character is not found.
DIR=	FORWARD (the default), to search from left to right. REVERSE, to search from right to left.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) The FIND instruction searches the first 20 bytes of MSG1 for the character '$'. If it finds a $, it stores the address of the character in the data area labeled POINTER. If the instruction does not find a $, it passes control to the instruction at label NOTFOUND. The direction of search is from left to right.

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

FIND - Locate a character *(continued)*

2) The FIND instruction searches for the string X'05' beginning at the address contained in index register 1. The search continues for the length value stored in the data area labeled LSTR. If the instruction finds the X'05' string, it stores the address of the string in the data area labeled POINTER. If the instruction does not find the string, it passes control to the instruction at label NOGOOD. The direction of the search is left to right.

```
          FIND    X'05',(0,#1),LSTR,POINTER,NOGOOD
```

Coding Example

To determine if a hyphen has been included in a 40-byte parts inventory number, the FIND instruction could be used as follows:

```
          .
          .
          .
GETPART# EQU      *
         READTEXT PARTNUM,'ENTER REQUESTED PART NUMBER',            X
               SKIP=1
*
FINDASH  EQU      *
         FIND     C'-',PARTNUM,40,POINTER,NOTVALID
         MOVEA    #1,PARTNUM                    GET PARTNUM ADDRESS
         SUBTRACT POINTER,#1,RESULT=LENGTH      FIND LENGTH OF PREFIX
         IF   (LENGTH,LE,1),GOTO,BADPREFX       IF FEWER THAN 2 REJECT IT
*
         IF   (LENGTH,LE,4),GOTO,GETCOST        IF FEWER THAN 5 IT'S OK
*
BADPREFX EQU      *                             ELSE REJECT IT
         PRINTEXT PARTNUM,SKIP=1
         PRINTEXT ' IS INVALID (PREFIX NOT OF ALLOWABLE SIZE)'
         GOTO     GETPART#                      RETRY
*
NOTVALID EQU      *
         PRINTEXT PARTNUM,SKIP=1
         PRINTEXT ' IS INVALID (MISSING HYPHEN) - REENTER'
         GOTO     GETPART#                      RETRY
*
GETCOST  EQU      *
          .
          .
PARTNUM  TEXT     LENGTH=40                 TEXT BUFFER FOR PART #
POINTER  DATA     F'0'                      POINTER TO ADDR OF CHAR
LENGTH   DATA     F'0'                      LENGTH OF PART # PREFIX
```

If the part number entered was 1213-9234, and the label PARTNUM was at address X'2040', the instruction would place a result of X'2044' in the data area labeled POINTER. The data area labeled LENGTH would contain a value of 4, and the program would branch to the label GETCOST.

FINDNOT - Locate the first different character

The FINDNOT instruction searches a character string for the first occurrence of a character (byte) that is different from the character you specify.

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
```

Operand	*Description*
character	FINDNOT searches for a character that is different from the one you specify for this operand. You can specify a text character or a a hexadecimal value.
string	The label of the string to be searched. The search will begin at the address of the label.
length	The number of bytes to be searched. You can code a positive integer or the label of a data area containing a positive integer.
where	The label of a data area where the address of the first different character is to be stored if it is found. If a different character is not found, this data area remains unchanged.
notfound	The label of the instruction to be executed if a different character is not found.
DIR=	FORWARD (the default), to search from left to right. REVERSE, to search from right to left.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) The FINDNOT instruction searches for the first nonblank character, starting at label INPUT. The search continues for 80 bytes. If a nonblank character is found, the character's address is stored in the data area labeled CPOINTER. If no characters are found during the 80-byte search, the FINDNOT instruction passes control to the instruction at label ALLBLANK. The direction of the search is from left to right.

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

FINDNOT - Locate the first different character *(continued)*

2) This instruction searches for the first bit string other than X'40'. The search starts at label CARD+79 and continues for 80 bytes. If a bit string other than X'40' is found, the address of the bit string is stored in the data area labeled LASTCHAR. If no bit string other than X'40' is found during the search, the FINDNOT instruction passes control to the instruction at label ALLBLANK. The direction of search is from right to left.

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

Coding Example

To reduce fixed-length, 80-byte records to variable-length records, the FINDNOT instruction could be used as follows:

```
         .
         .
NEXTCARD EQU      *
         ADD      CARDNUM,1
         .
         .
         .
FINDLAST EQU      *
         FINDNOT  X'40',CARD+79,80,POINTER,BLANKCRD,                  X
               DIR=REVERSE
*
GOTCHAR  EQU      *
         MOVEA    #1,CARD          GET ADDRESS CARD BUFFER
         SUBTRACT POINTER,#1,                                         X
               RESULT=LENGTH       GET NOMINAL LENGTH
         ADD      LENGTH,1         BUMP TO TRUE LENGTH
         MOVE     (0,#2),LENGTH    STORE LENGTH OF DATA
         ADD      #2,2             BUMP BUFFER POINTER
         MOVE     (0,#2),CARD,(1,BYTES),                              X
               P3=LENGTH           STORE CARD DATA
         ADD      #2,LENGTH        BUMP BUFFER BY DATA SIZE
         GOTO     NEXTCARD         GET ANOTHER CARD
*
BLANKCRD EQU      *
         PRINTEXT ' CARD # '       PRINT MESSAGE ON
         PRINTNUM CARDNUM          LISTING INDICATING THAT
*                                  THE CARD WAS BLANK
         PRINTEXT ' IS REJECTED AS BLANK'
         ADD      BLANKS,1         INCR. BLANK CARD COUNT
         GOTO     NEXTCARD         GET ANOTHER CARD
*
CARDNUM  DATA     F'0'                    CARDS READ COUNTER
POINTER  DATA     F'0'                    POINTER TO ADDR OF CHAR
CARD     DATA     CL80' '                 STORAGE BUFFER
BLANKS   DATA     F'0'                    BLANK CARD COUNTER
         .
         .
```

If the data on the card occupied the first 15 character positions and the next available buffer location (indexed by register #2) was X'5C00', POINTER would return as X'5C0E'. LENGTH would compute as X'000F' (X'000E' + X'0001'). Locations X'5C00'-X'5C01' would contain X'000F' and addresses X'5C02' through X'5C10' would receive the data. Register #2 would then be set to X'5011' and another card would be searched.

FIRSTQ - Acquire the first queue entry in a chain

The FIRSTQ instruction acquires the first (oldest) entry in a queue. You define a queue with the DEFINEQ statement. A queue entry can contain data or the address of a data buffer.

When you acquire the oldest entry with the FIRSTQ instruction, the second oldest entry becomes the first or oldest entry in the queue. After you acquire the contents of the oldest entry, the system adds the entry to the free chain of the queue.

Syntax:

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

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

Operand	*Description*
qname	The name of the queue from which the entry is to be fetched. The queue name is the label of the DEFINEQ statement that creates the queue.
loc	The label of a word of storage where the entry is placed. You can use the index registers, #1 and #2.
EMPTY=	The first instruction of the routine to be invoked if a "queue empty" condition is detected during the execution of this instruction. If you do not specify this operand, control returns to the next instruction after the FIRSTQ. A return code of -1 in the first word of the task control block indicates that the operation completed successfully. A return code of +1 indicates that the queue is empty.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

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

FIRSTQ - Acquire the first queue entry in a chain *(continued)*

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
1	Queue is empty

FMULT - Multiply floating-point values

The floating-point multiply instruction (FMULT) multiplies a floating-point value in operand 1 by a floating-point value in operand 2. You can use positive or negative values.

You must code FLOAT=YES on the PROGRAM statement of a program that uses floating-point instructions in its initial task 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
```

Operand	*Description*
opnd1	The label of the data area containing the value multiplied by opnd2. Opnd1 cannot be a self-defining term. The system stores the result of the operation in opnd1 unless you code the RESULT operand.
opnd2	The value by which opnd1 is multiplied. You can specify a self-defining term or the label of a data area. The valid range for this operand is from -32768 and +32767.
RESULT=	The label of a data area in which the result is placed. When you specify RESULT, the value of opnd1 does not change during the operation.
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 the RESULT operand was coded. Permissible characters are: F - Single-precision (32 bits) L - Extended-precision (64 bits) * - Default (single-precision) The default is single-precision.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

FMULT - Multiply floating-point values *(continued)*

Index Registers

You cannot use the index registers (#1 and #2) as operands in floating-point operations because they are only 16 bits in length. You can, however, use the software registers to specify the address of a floating-point operand.

Syntax Examples

1) The FMULT instruction multiplies two single-precision floating-point values and stores the result in RESULTF.

```
FLOAT     PROGRAM    START,FLOAT=YES
            .
            .
          FMULT      OP1F,OP2F,RESULT=RESULTF,PREC=FFF
            .
            .
            .
OP1F      DC         E'1.5'
OP2F      DC         E'0.2'
RESULTF   DC         E'0'
```

After the FMULT operation, RESULTF contains the value .30 .

2) The FMULT instruction multiplies two extended-precision floating-point values and stores the result in RESULTL.

```
FLOAT     PROGRAM    START,FLOAT=YES
            .
            .
          FMULT      OP1L,OP2L,RESULT=RESULTL,PREC=LLL
            .
            .
            .
OP1L      DC         L'50000.5'
OP2L      DC         L'40.4'
RESULTL   DC         L'0'
```

After the FMULT operation, RESULTL contains the value 2020020.20 .

FMULT - Multiply floating-point values *(continued)*

3) The FMULT instruction multiplies two single-precision floating-point values written in exponent (E) notation. The result is stored in RESULTFE.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FMULT     OP1FE,OP2FE,RESULT=RESULTFE,PREC=FFF
            .
            .
            .
OP1FE     DC        E'2.5E+1'        Equals decimal 25.0
OP2FE     DC        E'0.5E-1'        Equals decimal .05
RESULTFE  DC        E'0'
```

After the FMULT operation, RESULTFE contains the value .1250E+01 . This value is equal to the decimal value 1.250 .

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). You must test for the return code 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

FORMAT

FORMAT - Format data for display or storage

The FORMAT statement 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 referred to and cannot be placed within a sequence of executable instructions.

Note: The FORMAT statement can 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 cannot be used to continue a line before column 71.

Syntax:

```
label          FORMAT    (list),gen

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

Operand	*Description*
list	The format you want the data to be in after it is converted. The valid options are:

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 instruction.

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.

FORMAT - Format data for display or storage *(continued)*

The PUTEDIT instruction 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 self-defining terms can be inserted.

The GETEDIT instruction 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 instruction 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 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.

FORMAT - Format data for display or storage *(continued)*

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 is **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. Positive values do not require a position for the sign. If the number has fewer 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 quantity 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 is **Fw.d**.

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

If insufficient positions are reserved by "d", the number is rounded upwards. 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 fewer 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.

FORMAT - Format data for display or storage *(continued)*

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. A "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 two or four words of storage.

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

4. Numbers for F-conversion input need not have the decimal point appearing in the input field (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 differs 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 is **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 enough 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.

FORMAT - Format data for display or storage *(continued)*

The following examples show how each value 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. A "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 the decimal point appearing in the input field (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 differs 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

FORMAT - Format data for display or storage *(continued)*

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 a program does not change, such as printed headings.

The A-specification is used for alphameric data in storage that a program operates on, 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 is **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:

```
FORMAT ('DO && DON''T')
```

Literal data can be used only in loading a text buffer; it is invalid in a GETEDIT instruction. 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
```

FORMAT - Format data for display or storage *(continued)*

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 is **Aw**.

The specification Aw is used to transmit alphameric data to or from data areas 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 ellipsis (....) represents 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 before data conversion. The alphameric data is left justified in the field.

Blank Specification: General form is **X**.

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

FORMAT - Format data for display or storage *(continued)*

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)
```

can 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

You can insert blank lines between output records by using consecutive slashes (/). The slash causes a line-control character to be inserted into the buffer. The number of blank lines inserted between output records depends on 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)
```

FORMAT - Format data for display or storage *(continued)*

Repetitive Specification

You can repeat a specification, within the limits of the text buffer size, by coding an integer from 1 to 255 before the specification.

For example,

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

is equivalent to:

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

and uses less storage.

You can use a parenthetical expression with a multiplier (repeat constant) to repeat data fields according to the format specifications contained within the parentheses. All item types are permitted within the parenthetical expression except another parenthetical expression. You can specify multiple parenthetical expressions 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)
```

FORMAT - Format data for display or storage *(continued)*

Storage Considerations

In general, the fewer items in the FORMAT list, the less storage required. An item is defined as a single conversion specification, a 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)
```

Coding Example

The following example begins by executing a PRINTEXT instruction that prints a message requesting the model year and serial numbers for the automobile of interest. The first GETEDIT actually reads the two requested numbers into a TEXT statement labeled TEXT1.

The GETEDIT instruction searches the TEXT1 data and converts the first entry to a single-precision variable called LIST1. The second entry is converted to a double-precision variable called LIST2. Both LIST1 and LIST2 are then converted back to EBCDIC and displayed on the printer by the first PUTEDIT instruction using the PE1FMT FORMAT statement. The PUTEDIT instruction and FORMAT statement determine the layout of the data as it is displayed.

The GETEDIT instruction following label GE2 takes the data already entered into TEXT1 with the preceding READTEXT and again converts it into the two binary variables called LIST1 (single-precision) and LIST2 (double-precision). Because ACTION=STG, a READTEXT must be issued before executing the GETEDIT.

The PUTEDIT instruction at label PE2 converts the two variables back to EBCDIC and places them into the TEXT2 statement as formatted by the PE2FMT FORMAT statement. Again, the keyword ACTION=STG prevents the data from being printed until the following PRINTEXT instruction is executed.

FORMAT - Format data for display or storage *(continued)*

```
GE1      EQU      *
         PRINTEXT '@ENTER MODEL YEAR AND SERIAL NUMBER@'
         GETEDIT  GE1FMT,TEXT1,(LIST1,(LIST2,D)),                     X
               ACTION=IO,ERROR=ERR1
*
PE1      EQU      *
         ENQT     $SYSPRTR
         PUTEDIT  PE1FMT,TEXT2,(LIST1,(LIST2,D)),                     X
               ACTION=IO
*

GE2      EQU      *
         READTEXT TEXT1,'@ENTER YOUR DEPT. AND SYSTEM ID NUMBER@'
*
         GETEDIT  GE2FMT,TEXT1,(LIST1,(LIST2,D)),                     X
               ACTION=STG,ERROR=ERR1
*
PE2      EQU      *
         PUTEDIT  PE2FMT,TEXT2,(LIST1,(LIST2,D)),ACTION=STG
         ENQT     $SYSPRTR
         PRINTEXT TEXT2
         DEQT
          .
          .
          .
ERR1     EQU      *
         PRINTEXT '@GETEDIT GE1 HAS FAILED@'
         GOTO     ERROROUT
*
ERR2     EQU      *
         PRINTEXT '@GETEDIT GE2 HAS FAILED@'
         GOTO     ERROROUT
*
GE1FMT   FORMAT   (I4,1X,I8)
PE1FMT   FORMAT   ('MDL. YR. = ',I4,6X,:'SER. NO. = ',I8)
GE2FMT   FORMAT   (I3,1X,I6)
PE2FMT   FORMAT   ('DEPT. = ',I3,4X,'SYST. ID. = ',I6)
LIST1    DATA     F'0'
LIST2    DATA     D'0'
TEXT1    TEXT     LENGTH=13
TEXT2    TEXT     LENGTH=42
ERROROUT EQU      *
```

FPCONV - Convert to or from floating-point

The FPCONV instruction converts integer values to or from floating-point numbers by using the optional floating-point hardware feature.

You must code FLOAT=YES on the PROGRAM statement of programs whose primary 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
```

Operand	*Description*
opnd1	The label of the data area to receive the result of the conversion.
opnd2	The label of the data area that contains the value to be converted. You can also code an integer number between -32768 and +32767.
COUNT=	The number of values in opnd2 to be converted and stored at locations beginning at opnd1. If opnd2 is immediate data, it is converted and placed in the storage area defined by opnd1 in the number of consecutive locations defined by this operand.
PREC=xy	Defines the precision of opnd1 and opnd2 and the type of data (integer or floating-point) you coded for these operands. Specify the precision and data type in the form PREC=xy, where "x" is the precision and data type for opnd1 and "y" is the precision and data type for opnd2. Opnd1 and opnd2 cannot be the same data type. The valid precisions and data types for "x" and "y" are as follow: S - Single-precision integer (1 word) D - Double-precision integer (2 words) F - Single-precision floating-point value L - Extended-precision floating-point value * - Use default (single-precision)
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

FPCONV - Convert to or from floating-point *(continued)*

Syntax Examples

1) Convert five double-precision integers beginning at label B to extended-precision floating-point values. Store the result beginning at label A.

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

2) Convert an extended-precision floating-point value at label L4 to a double-precision integer. Store the result beginning at label X.

```
FPCONV  X,L4,PREC=DL
```

3) Convert a single-precision integer value at label C to a single-precision floating-point value. Store the result beginning at the indexed location (6,#1).

```
FPCONV  (6,#1),C
```

4) Convert an extended-precision floating-point value at the indexed location of (X,#1) to a double-precision integer. Store the result beginning at the indexed location (Y,#2).

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

FPCONV - Convert to or from floating-point *(continued)*

Coding Example

The example estimates the number of hours required for a plane, carrying a specified load weight, to travel to a destination a given number of miles from its departure point.

The FPCONV instruction at label FP1 converts a single-precision integer to single-precision floating-point value. This instruction uses the default precision.

The FPCONV instruction, at label FP2, converts a double-precision integer to a single-precision floating-point value.

At label FP3, the FPCONV instruction converts two single-precision integers to single-precision floating-point values. The values to be converted are indexed and the parameter naming operand (P1=) allows the result field locations to be assigned dynamically.

The FPCONV instruction at label FP4 converts a single-precision floating-point value to a single-precision integer.

```
CONVERT  PROGRAM  START,FLOAT=YES
START    EQU      *
         GETVALUE MILES,'@ENTER MILES TO DESTINATION'
FP1      FPCONV   FMILES,MILES
         GETVALUE FREIGHT,'@POUNDS OF CARGO ?',FORMAT=(10,0,I),TYPE=D
FP2      FPCONV   FFREIGHT,FREIGHT,PREC=FD
         READTEXT TYPE,'@ENTER PLANE TYPE'
         CALL     FINDTYPE,TYPE
         MOVEA    #1,BUFR
         MOVEA    RESULT,FFUELUSE
           .
FP3      FPCONV   *,(32,#1),COUNT=2,P1=RESULT
         CALL     CALCTIME
           .
FP4      FPCONV   ELAPSED,FELAPSED,PREC=SF
         PRINTEXT '@NUMBER OF HOURS OF ELAPSED FLIGHT TIME '
         PRINTNUM ELAPSED
           .
           .
BUFR     DATA     256H'0'
TYPE     TEXT     LENGTH=4
MILES    DATA     F'0'
FREIGHT  DATA     D'0'
ELAPSED  DATA     F'0'
*
FMILES   DATA     E'0'
FFREIGHT DATA     E'0'
FFUELUSE DATA     E'0'
FSPEED   DATA     E'0'
FELAPSED DATA     E'0'
           .
           .
```

FREESTG - Free mapped and unmapped storage areas

The FREESTG instruction releases the mapped and unmapped storage areas you obtained with the GETSTG instruction.

Note: "Mapped storage" is the physical storage you defined on the SYSTEM statement during system generation. "Unmapped storage" is any physical storage that you did not include on the SYSTEM statement.

Syntax:

```
label         FREESTG   name,TYPE=,ERROR=,P1=

Required:     name
Defaults:     TYPE=ALL
Indexable:    none
```

Operand	*Description*
name	The label of a STORBLK statement. The STORBLK statement defines the mapped and unmapped storage areas that your program uses.
TYPE=	ALL, the default, to release the mapped storage area and all the unmapped storage areas your program acquired with GETSTG instruction. UNMAP, to release only the unmapped storage areas your program acquired with the GETSTG instruction.
ERROR=	The label of the first instruction of the routine to be invoked if an error occurs during the execution of this instruction.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

FREESTG - Free mapped and unmapped storage areas *(continued)*

Syntax Examples

1) Release the mapped storage area and all unmapped storage areas defined by the STORBLK statement labeled BLOCK.

```
FREESTG     BLOCK
```

2) Release only the unmapped storage areas defined by the STORBLK statement labeled BLOCK.

```
FREESTG     BLOCK,TYPE=UNMAP
```

3) Release the mapped storage area and all unmapped storage areas defined by the STORBLK statement labeled BLOCK. The label of the first instruction of the error routine is OUT.

```
FREESTG     BLOCK,TYPE=ALL,ERROR=OUT
```

Coding Example

See the SWAP instruction for an example that uses the FREESTG instruction.

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
1	No storage entries exist in storage control block
2	Error occurred while freeing the mapped storage area
100	No unmapped storage support in system

FSUB - Subtract floating-point values

The floating-point subtract instruction (FSUB) subtracts a floating-point value in operand 2 from a floating-point value in operand 1. You can use positive or negative values.

You must code FLOAT=YES on the PROGRAM statement of a program that uses floating-point instructions in its initial task 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
```

Operand	*Description*
opnd1	The label of the data area from which opnd2 is subtracted. Opnd1 cannot be a self-defining term. The system stores the result of the operation in opnd1 unless you code the RESULT operand.
opnd2	The value subtracted from opnd1. You can specify a self-defining term or the label of a data area. The valid range for this operand is from -32768 to +32767.
RESULT=	The label of a data area in which the result is to be placed. When you specify RESULT, the value of opnd1 does not change during the operation.
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 the RESULT operand was coded. Permissible characters are: F - Single-precision (32 bits) L - Extended-precision (64 bits) * - Default (single-precision) The default is single-precision.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

FSUB - Subtract floating-point values *(continued)*

Index Registers

You cannot use the index registers (#1 and #2) as operands in floating-point operations because they are only 16 bits in length. You can, however, use the software registers to specify the address of a floating-point operand.

Syntax Examples

1) The FSUB instruction subtracts two single-precision floating-point values and stores the result in RESULTF.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FSUB      OP1F,OP2F,RESULT=RESULTF,PREC=FFF
            .
            .
            .
OP1F      DC        E'1.5'
OP2F      DC        E'0.2'
RESULTF   DC        E'0'
```

After the FSUB operation, RESULTF contains the value 1.30.

2) The FSUB instruction subtracts two extended-precision floating-point values and stores the result in RESULTL.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FSUB      OP1L,OP2L,RESULT=RESULTL,PREC=LLL
            .
            .
            .
OP1L      DC        L'50000.5'
OP2L      DC        L'40.4'
RESULTL   DC        L'0'
```

After the FSUB operation, RESULTL contains the value 49960.10.

FSUB - Subtract floating-point values *(continued)*

3) The FSUB instruction subtracts two single-precision floating-point values written in exponent (E) notation. The result is stored in RESULTFE.

```
FLOAT     PROGRAM   START,FLOAT=YES
            .
            .
          FSUB      OP1FE,OP2FE,RESULT=RESULTFE,PREC=FFF
            .
            .
            .
OP1FE     DC        E'2.5E+1'       Equals decimal 25.0
OP2FE     DC        E'0.5E-1'       Equals decimal .05
RESULTFE  DC        E'0'
```

After the FSUB operation, RESULTFE contains the value .2495E+02. This value is equal to the decimal value 24.95.

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). You must test for the return code 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

GETEDIT - Collect and store data

The GETEDIT instruction acquires data from a terminal or storage area, converts the data according to a FORMAT list, and stores the data in your program at the locations specified by the data list.

When you use the GETEDIT instruction in your program, you must link-edit your program using the "autocall" option of $EDXLINK. Refer to the *Event Driven Executive Language Programming Guide* for information on how to link-edit programs.

The supervisor places a return code in the first word of the task control block (taskname) whenever a GETEDIT instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

See Figure 8 on page LR-217 for an illustration of how the GETEDIT instruction works.

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
```

Operand	***Description***
format	The label of a FORMAT statement or the label to be attached to the format list optionally included in this statement. 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 label of a TEXT statement defining a storage area for character data. If data is moved from a terminal, this area 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,...)

GETEDIT - Collect and store data *(continued)*

or

((variable,count),...)

or

((variable,type),...)

where:

variable is the label 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. The type can be:

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

The type defaults to S for integer format data and to F for floating-point format data.

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

ERROR= The label of the routine to receive control if the system detects an error during the GETEDIT operation. The system returns a return code to the task even if you do not code this operand.

Errors that might cause the system to invoke the error routine are:

- Use of an incorrect format list
- Field omitted (attempt is made to convert the rest)
- Not enough data in input text buffer to satisfy the data list
- Conversion error (value too large).

GETEDIT - Collect and store data *(continued)*

ACTION= IO (the default), causes a READTEXT instruction to be executed before 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 system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

GETEDIT - Collect and store data *(continued)*

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

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.

The PROTECT operand does not apply to the 3101 in block mode.

3101 Display Considerations

When using a 3101 in block mode, the attribute byte associated with the prompt message and the input data will depend on the current TERMCTRL SET,ATTR in effect. The default is SET,ATTR=HIGH (high intensity) for the attribute byte.

Syntax Examples

1) The following GETEDIT instruction converts the first four characters to an integer and stores them at A. It converts the next six characters to a single-precision floating-point value and stores them at B. The next two characters are bypassed, and the last 10 characters are converted to an extended-precision floating-point value (because of the E-type specification) and are stored at C.

```
         GETEDIT   FM,TEXT1,(A,(B,F),(C,L))
            .
            .
            .
TEXT1    TEXT      LENGTH=24
FM       FORMAT    (I4,F6.2,2X,E10.4)
```

GETEDIT - Collect and store data *(continued)*

2) This GETEDIT instruction converts four integer values contained in the text buffer XSCREEN to a single hexadecimal word. The GETEDIT instruction places the results in the location SCREEN.

```
         GETEDIT  FM1,XSCREEN,((SCREEN,S)),ACTION=STG
            .
            .
            .
FM1      FORMAT   (I4),GET
XSCREEN  TEXT     LENGTH=4
```

Coding Example

The example begins by executing a PRINTEXT instruction that issues a message requesting the model year and serial numbers for the automobile of interest. The first GETEDIT actually reads the two requested numbers with a TEXT statement labeled TEXT1.

The GETEDIT instruction searches the TEXT1 data and converts the first entry to a single-precision variable called LIST1. The second entry is converted to a double-precision variable called LIST2. The first PUTEDIT instruction, using the FORMAT statement labeled PE1FMT, converts LIST1 and LIST2 back to EBCDIC and displays these values on the printer. The PUTEDIT instruction and FORMAT statement determine the layout of the data as it is displayed.

The GETEDIT instruction after label GE2 takes the data already entered into TEXT1 with the preceding READTEXT and converts it into the two binary variables called LIST1 (single-precision) and LIST2 (double-precision). Because ACTION=STG, a READTEXT must be issued before executing the GETEDIT.

The PUTEDIT instruction at label PE2 converts the two variables back to EBCDIC and places them into the TEXT2 statement as formatted by the PE2FMT FORMAT statement. Again, the keyword ACTION=STG prevents the data from being printed until the following PRINTEXT instruction is executed.

GETEDIT - Collect and store data *(continued)*

```
GE1       EQU      *
          PRINTEXT '@ENTER MODEL YEAR AND SERIAL NUMBER@'
          GETEDIT  GE1FMT,TEXT1,(LIST1,(LIST2,D)),                 X
                   ACTION=IO,ERROR=ERR1
*
PE1       EQU      *
          ENQT     $SYSPRTR
          PUTEDIT  PE1FMT,TEXT2,(LIST1,(LIST2,D)),ACTION=IO
          DEQT
*
GE2       EQU      *
          READTEXT TEXT1,'@ENTER YOUR DEPT. AND SYSTEM ID NUMBER@'
*
          GETEDIT  GE2FMT,TEXT1,(LIST1,(LIST2,D)),                 X
                ACTION=STG,ERROR=ERR1
*
PE2       EQU      *
          PUTEDIT  PE2FMT,TEXT2,(LIST1,(LIST2,D)),ACTION=STG

          ENQT     $SYSPRTR
          PRINTEXT TEXT2
          DEQT
           .
           .
           .
ERR1      EQU      *
          PRINTEXT '@GETEDIT GE1 HAS FAILED@'
          GOTO     ERROROUT
*
ERR2      EQU      *
          PRINTEXT '@GETEDIT GE2 HAS FAILED@'
          GOTO     ERROROUT
GE1FMT    FORMAT   (I4,1X,I8)
PE1FMT    FORMAT   ('MDL. YR. = ',I4,6X,'SER. NO. = ',I8)
GE2FMT    FORMAT   (I3,1X,I6)
PE2FMT    FORMAT   ('DEPT. = ',I3,4X,'SYST. ID. = ',I6)
LIST1     DATA     F'0'
LIST2     DATA     D'0'
TEXT1     TEXT     LENGTH=13
TEXT2     TEXT     LENGTH=42
ERROROUT  EQU      *
```

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

For several errors, the system returns the return code with the highest value.

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

GETEDIT - Collect and store data *(continued)*

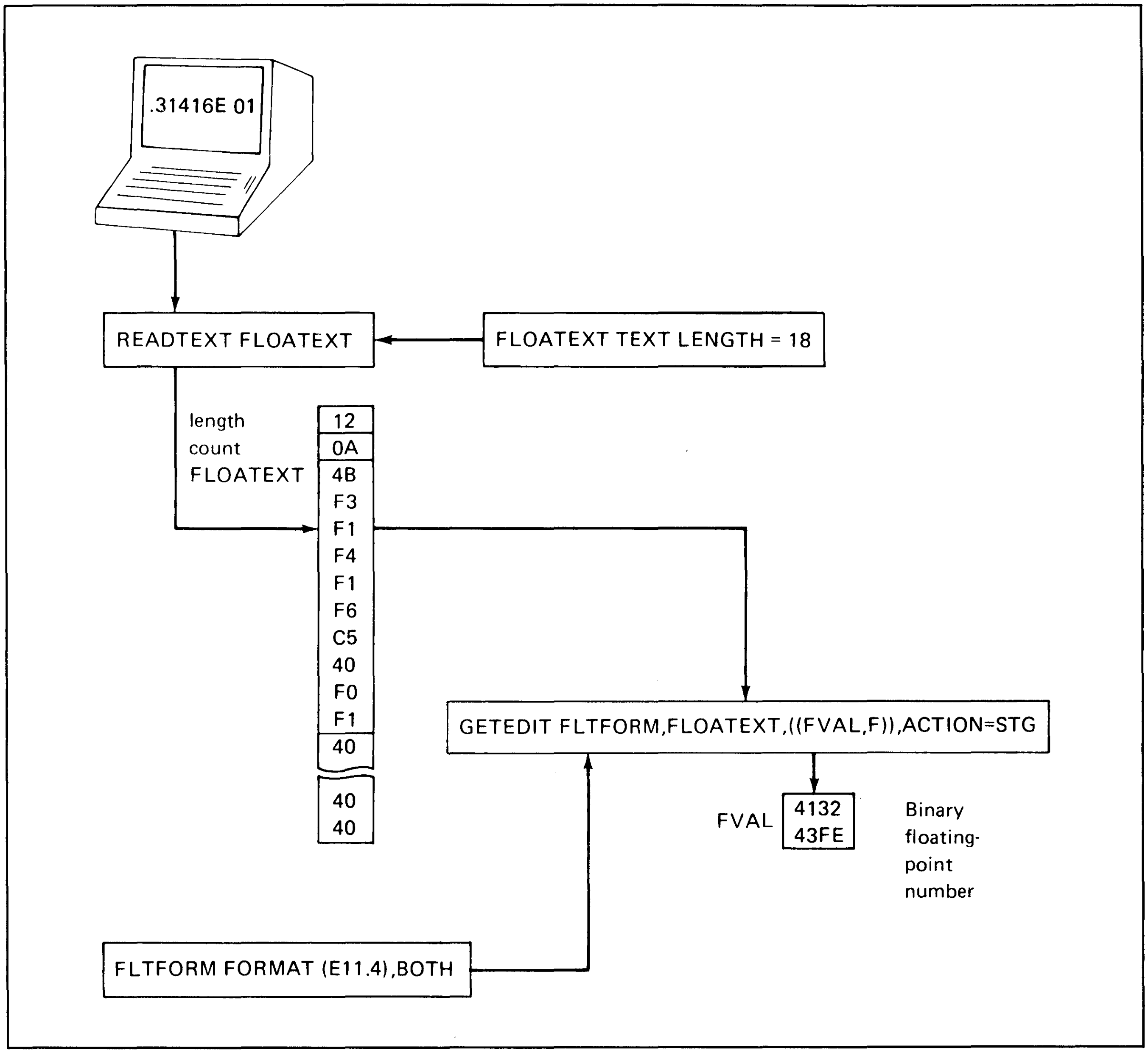

Figure 8. GETEDIT Overview

GETSTG

GETSTG - Obtain mapped and unmapped storage areas

The GETSTG instruction obtains mapped and unmapped storage areas.

The SWAP instruction allows your program to use the unmapped storage areas you acquire with the GETSTG instruction. You release mapped and unmapped storage areas with the FREESTG instruction.

Note: "Mapped storage" is the physical storage you defined on the SYSTEM statement during system generation. "Unmapped storage" is any physical storage that you did not include on the SYSTEM statement.

Syntax:

```
label         GETSTG     name,TYPE=,ERROR=,P1=

Required:     name
Defaults:     TYPE=ALL
Indexable:    none
```

Operand	*Description*
name	The label of a STORBLK statement. The STORBLK statement specifies the size of the mapped storage area and the number of unmapped storage areas the GETSTG instruction can obtain.
TYPE=	MAP, to acquire only the mapped storage area you defined on the STORBLK statement. NEXT, to acquire one of the unmapped storage areas you defined on the STORBLK statement. The instruction also obtains the mapped storage area if it has not acquired it already. ALL, the default, to acquire all the unmapped storage areas you defined on the STORBLK statement. The instruction also obtains the mapped storage area if it has not acquired it already.
ERROR=	The label of the first instruction of the routine to be invoked if an error occurs during the execution of this instruction.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

GETSTG - Obtain mapped and unmapped storage areas *(continued)*

Syntax Examples

1) Obtain all the unmapped storage areas and the mapped storage area defined on the STORBLK statement labeled BLOCK.

```
GETSTG    BLOCK,TYPE=ALL
```

2) Obtain only the mapped storage area defined on the STORBLK statement labeled BLOCK.

```
GETSTG    BLOCK,TYPE=MAP
```

3) Obtain one of the unmapped storage areas defined on the STORBLK labeled BLOCK. The label of the first instruction of the error routine for this instruction is OUT.

```
GETSTG    BLOCK,TYPE=NEXT,ERROR=OUT
```

Coding Example

See the SWAP instruction for an example that uses the GETSTG instruction.

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion.
1	A mapped storage entry already exists in the storage control block.
2	Mapped storage area is not available in the system.
100	No unmapped storage support in system
3	Unmapped storage is not available or only partial storage was obtained. Check the second word of the TCB. A zero shows that no unmapped storage is available. A nonzero value equals the number of unmapped storage areas obtained by the instruction.
4	All unmapped storage entries in the storage control block are in use.

GETTIME - Get date and time

The GETTIME instruction places the contents of the system's time-of-day clock in a 3-word table that you define in your program. The 3 words contain the hours, minutes, and seconds, in that order. You also can specify that the date be stored in an additional 3 words, resulting in a 6-word table containing hours, minutes, seconds, month, day, and year. Use this instruction when you want to store the time of day and date as you collect data.

The maximum time on the clock is 23.59.59. At midnight, the supervisor resets the time-of-day clock to 0 and increases the date by 1. The supervisor resets the month and year as necessary.

Syntax:

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

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

Operand	*Description*
loc	The label of a 3-word table where the system stores the time of day as hours, minutes, and seconds; or the label of a 6-word table where the time of day and the date are stored as hours, minutes, seconds, month, day, and year. The time and date are in hexadecimal format.
DATE=	YES, to obtain the date as well as the time of day. If the task control block code word, $TCBCO, contains a -2, the date is in the form: day, month, year. If $TCBCO contains a -1, the date is in the form: month, day, year. The format of the date was specified on the SYSTEM statement during system generation.
	NO, to obtain only the hours, minutes, and seconds, in that order.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

GETTIME - Get date and time *(continued)*

Syntax Example

This GETTIME instruction obtains the time and date and places the result in a 6-word table beginning at the label TAB.

```
GETTIME   TAB,DATE=YES
```

The following example shows the possible contents of TAB (in hexadecimal format) after the GETEDIT operation:

TAB	000D	(hours)
	0018	(minutes)
	0005	(seconds)
	0007	(month)
	001B	(day)
	0053	(year)

The time and date shown is 13:24:05 on July 27, 1983.

Coding Example

The following program demonstrates a method of acquiring the system date and time then displaying both on a terminal according to the coded FORMAT statement.

```
DTERTN    PROGRAM   START
START     EQU       *
          ENQT      $SYSLOG
          GETTIME   TAB,DATE=YES
          PUTEDIT   FORMAT,TEXT,((TAB,6,S)),LINE=8,ERROR=ERR
          GOTO      DONE
*
ERR       EQU       *
          IF        DTERTN+2,NE,-1
          MOVE      CODE,DTERTN+2
          PRINTEXT  '@RETURN CODE: '
          GOTO      DONE
          ENDIF
*
DONE      EQU       *
          DEQT
          PROGSTOP
CODE      TEXT      LENGTH=2
TAB       DATA      6F'0'
TEXT      TEXT      LENGTH=36
FORMAT    FORMAT    ('TIME ',I2,':',I2,':',I2,10X,                      X
               'DATE ',I2,'/',I2,'/',I2)
          ENDPROG
          END
```

GETVALUE - Read a value entered at a terminal

The GETVALUE instruction reads one or more integer values, or a single floating-point value, entered at a terminal. The values can be decimal or hexadecimal, and of single or double precision. The system treats invalid characters as delimiters.

The supervisor places a return code in the first word of the task control block (taskname) whenever a GETVALUE instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

Syntax:

```
label         GETVALUE loc,pmsg,count,MODE=,PROMPT=,
                       FORMAT=,TYPE=,SKIP=,LINE=,SPACES=,
                       COMP=,PARMS=(parm1,...,parm8),
                       MSGID=,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,MSGID=NO
Indexable:    pmsg,SKIP,LINE,SPACES
```

Operand	***Description***
loc	The label of the variable to receive the input value. If your program requests more than one value, the system stores the successive values in successive words or doublewords depending on the precision you specify in the count operand.
pmsg	The prompt message. Code the label of a TEXT statement or an explicit text message enclosed in single quotes. The GETVALUE instruction issues this prompt according to the parameter you code for the PROMPT keyword. To retrieve a prompt message from a data set or module containing formatted program messages, code the number of the message you want displayed or printed. You must code a positive integer or a label preceded by a plus sign (+) that is equated to a positive integer. If you retrieve a prompt message from storage, you must also code the COMP= operand. See Appendix E, "Creating, Storing, and Retrieving Program Messages" on page LR-615 for more information.
count	The number of integer values to be entered. If the FORMAT parameter is used, the count is forced to 1 regardless of the value specified. The precision specification can be substituted for the count specification. If the precision is substituted for the count, the count defaults to 1. The precision can accompany the count in the form of a sublist: (count,precision). The default value for

GETVALUE - Read a value entered at a terminal *(continued)*

precision is word, or the keyword WORD can be specified. If double-precision is desired, code the precision keyword DWORD. Only the WORD and DWORD precisions can be specified.

With conditional prompting, the system issues the prompt message if you do not enter advance input. Once a prompt message has been issued, however, you may enter one or more values. Omitted values leave the corresponding internal variables unchanged and are indicated by coding two consecutive delimiters. The delimiters allowed between values are the characters slash (/), comma (,), period (.), or blank ( ). The number of values entered is stored at taskname+2 when the instruction completes.

MODE= HEX, for hexadecimal input.

DEC, the default, for decimal input.

PROMPT= COND (conditional), to prevent the system from displaying the prompt message if you enter a value before the prompt.

UNCOND (unconditional), to have the system display the prompt message without exception. UNCOND is the default.

FORMAT= The format of the value to be read in. Use the FORMAT operand where the default is not desired. 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 expected from the terminal. Count the decimal point as part of the field width.

d A decimal value equal to the number of digits 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. Code I for integer data, F for floating-point data (XXXX.XXX), or E for floating-point data in E notation. See the value operand under the DATA/DC statement for a description of E notation format.

Note: You can use the floating-point format for data even if you do not have floating-point hardware installed in your system. Floating-point hardware is required, however, to do floating-point arithmetic.

The first FORMAT operand to execute generates a work area which all subsequent FORMAT operands will use also. The generated work area is nonreentrant in a multitasking environment, and all tasks must use the ENQ/DEQ functions to serialize access to it.

GETVALUE - Read a value entered at a terminal *(continued)*

Note: If you code the FORMAT parameter and you are entering advanced input (PROMPT=COND) for multiple GETVALUE statements, a blank must be used to separate the input values. No other delimiters are valid.

TYPE= The type of variable to receive the input. Use this operand with FORMAT= only. The valid types are:

- 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 system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

GETVALUE - Read a value entered at a terminal *(continued)*

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

COMP= The label of a COMP statement. You must specify this operand if the GETVALUE instruction is retrieving a prompt message from a data set or module containing formatted program messages. The COMP statement provides the location of the message. (See the COMP statement for more information.)

PARMS= The labels of data areas containing information to be included in a message you are retrieving from a data set or module containing formatted program messages. You can code up to eight labels. If you code more than one label, you must enclose the list in parentheses.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

MSGID= YES, if you want the message number and four-character prefix to be printed at the beginning of the message you are retrieving from a data set or module containing formatted program messages. See the COMP statement operand 'idxx' for a description of the four-character prefix.

NO (the default), to prevent the system from printing or displaying this information at the beginning of the message.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

3101 Display Considerations

When using a 3101 in block mode, the attribute byte associated with any prompt message and the input data will depend on the current TERMCTRL SET,ATTR in effect. The default is SET,ATTR=HIGH (high intensity) for the attribute byte. Also TERMCTRL SET,STREAM=NO should be in effect when the GETVALUE instruction is executed for a 3101 in block mode.

GETVALUE - Read a value entered at a terminal *(continued)*

Syntax Examples

The syntax examples for this instruction use the following data areas:

```
MSG      TEXT    'ENTER NEXT NUMBER'
A        DC      F'0'
B        DC      F'0'
C        DC      F'0'
D        DC      D'0'
E        DC      D'0'
F        DC      E'0.0000'
L        DC      L'0.000'
```

1) Read a single-precision integer of up to six decimal digits into data area A.

```
GETVALUE    A,MSG

GETVALUE    A,MSG,TYPE=S,FORMAT=(6,0,I)
```

2) Read three consecutive single-precision integers (of six decimal digits or fewer) into data areas A, B, and C.

```
GETVALUE    A,MSG,(3,WORD)
```

3) Read a double-precision integer of up to 10 decimal digits into doubleword data area D.

```
GETVALUE    D,MSG,DWORD

GETVALUE    D,MSG,TYPE=D,FORMAT=(10,0,I)
```

4) Read two consecutive single-precision integers (of six decimal digits or fewer) into data areas B and C.

```
GETVALUE    B,MSG,2
```

5) Read two consecutive double-precision integers (of ten decimal digits or fewer) into data areas D and E.

```
GETVALUE    D,MSG,(2,DWORD)
```

6) Ignore the count and read a single-precision integer of up to four decimal digits into data area A.

```
GETVALUE    A,MSG,3,TYPE=S,FORMAT=(4,0,I)
```

GETVALUE - Read a value entered at a terminal *(continued)*

7) Read a double-precision integer of up to six decimal digits into doubleword data area E.

```
GETVALUE    E,MSG,TYPE=D,FORMAT=(6,0,I)
```

8) Read a single-precision floating-point (F-format) number of seven digits, with four digits to the right of an assumed decimal point, into data area F.

```
GETVALUE    F,MSG,TYPE=F,FORMAT=(8,4,F)
```

9) Read an extended-precision floating-point (E-format) number of eight digits, with three digits to the right of an assumed decimal point, into data area E.

```
GETVALUE    G,MSG,TYPE=L,FORMAT=(9,3,E)
```

Coding Examples

1) If, in the following example, the operator entered 55 23A5 68 in response to the prompt from the third GETVALUE, the first three of five storage locations in DATA3 would assume the values 0055, 23A5, and 0068, respectively. The other two word locations would remain unchanged (X'0000').

```
                .
          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'
                .
                .
```

GETVALUE - Read a value entered at a terminal *(continued)*

2)In the following example, the GETVALUE instruction, at label G1, prints a message then reads a value entered by an operator. Note that the message in single quotes is printed and provides an unconditional prompt. Also, the value read uses the following defaults: decimal, integer, 1 - 6 digits, and single-precision.

The GETVALUE at G2 issues a prompt only if there is no advance input and it reads 1 hexadecimal input value. Default values are in effect for the FORMAT and TYPE parameters.

The GETVALUE at G3 reads a variable number of hexadecimal input values, using the default FORMAT and TYPE parameters.

The G4 GETVALUE uses the FORMAT parameter to read a single, floating-point value of up to 9 digits in length and then places the result in a doubleword field.

```
              .
              .
              .
G1      GETVALUE COUNT,'@ HOW MANY WORDS OF STORAGE ? '
G2      GETVALUE DATA,'@ ENTER START ADDRESS',MODE=HEX,PROMPT=COND
        MOVE     #1,DATA
        AND      #1,X'FFFE'            INSURE EVEN STORAGE ADDRESS
        PRINTEXT '@ CURRENT VALUE(S) NOW :'
        PRINTNUM (0,#1),1,MODE=HEX,P2=COUNT
        MOVE     KOUNT,COUNT
G3      GETVALUE DATA,'@ ENTER NEW VALUE(S)',1,P3=KOUNT,MODE=HEX
              .
              .
G4      GETVALUE FLOAT,'@ ENTER DATA',FORMAT=(9,2,F),TYPE=D
              .
              .
              .
```

3) In this example, the GETVALUE instruction displays a prompt message contained in the disk data set MSGSET on volume EDX002. Because +MSG9 equals 9, the system retrieves the ninth message in MSGSET.

```
SAMPLE   PROGRAM    START,200,DS=((MSGSET,EDX002))
            .
         GETVALUE   PNUMB,+MSG9,PROMPT=COND,COMP=MSGSTMT
            .
MSG9     EQU        9
PNUMB    DATA       F'0'
MSGSTMT  COMP       'SRCE',DS1,TYPE=DSK
```

GETVALUE - Read a value entered at a terminal *(continued)*

Message Return Codes

The system issues the following GETVALUE return codes when you retrieve a prompt message from a data set or module containing formatted program messages. The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Message successfully retrieved
301-316	Error while reading message from disk. Subtract 300 from this value to get the actual return code. See the disk return codes following the READ or WRITE instruction for a description of the code.
326	Message number out of range
327	Message parameter not found
328	Instruction does not supply message parameter(s)
329	Invalid parameter position
330	Invalid type of parameter
331	Invalid disk message data set
332	Disk message read error
333	Storage resident module not found
334	Message parameter output error
335	Disk messages not supported (MINMSG support only)

GIN - Enter unscaled cursor coordinates

The GIN instruction allows you to specify unscaled cursor coordinates interactively. The instruction rings the bell, displays cross-hairs, and waits for you to position the cross-hairs and enter a single character. GIN then stores the coordinates of the cross-hair cursor. It also stores the character you entered, if you request this.

Cursor coordinates are unscaled. The PLOTGIN instruction obtains coordinates scaled by the use of a PLOTCB control block.

Syntax:

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

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

Operand	*Description*
x	The location where the x cursor coordinate value is to be stored.
y	The location where the y cursor coordinate value is to be stored.
char	The location where the character you select is to be stored. The character is stored in the right-hand byte. The left byte is set to zero. If you do not code this operand, the instruction does not store the selected character.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Store the x coordinate in X and the y coordinate in Y. Store the character in the location CHAR.

```
GIN     X,Y,CHAR
```

GOTO - Go to a specified instruction

The GOTO instruction allows you to pass control, or "branch," to another instruction in the program.

The statement can:

- Pass control directly to the label of an instruction.
- Pass control to an address defined by a label.
- Pass control to one of the labels in a list based on the value of an index word.

GOTO can also be used as an operand of the IF instruction.

Syntax:

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

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

Operand	***Description***
loc	The label of the instruction to receive control. Enclose this label in parentheses if the label points to a data area containing the address of the next instruction to be executed. It may also be a displacement value from index register #1 or #2. The instruction you branch to must be on a fullword boundary.
loc0,loc1, ...,locn	The labels in a list of instruction labels that can receive control depending on the value of the index word. The label at loc1 receives control if the index value is equal to 1. The label at loc2 receives control if the index value is equal to 2, and so on. The first label, loc0, is the label of the instruction that receives control if the value of the index word is not in the range of loc1-locn. The number of instruction labels in the list plus 1 must not exceed 50.
index	The label of an index word containing a value that determines the label to branch to in a list of labels.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

GOTO - Go to a specified instruction *(continued)*

Syntax Examples

1) Branch to the label EXIT.

```
        GOTO    EXIT
```

2) Move the address of the ADD instruction into HOLD and branch to that address.

```
        MOVEA   HOLD,NEXT
          .
          .
        GOTO    (HOLD)
          .
          .
          .
NEXT    ADD     A,B
          .
          .
          .
HOLD    DATA    F'0'
```

3) The branch depends on the value in INDEX. If the value in INDEX is 1, the instruction branches to label L1. If the value in INDEX is 2, the instruction branches to label L2. Any other value in INDEX causes the instruction to branch to ERR.

```
        GOTO  (ERR,L1,L2),INDEX
```

Another example using GOTO is shown under "Syntax Examples with IF, ELSE, and ENDIF" on page LR-239.

HASHVAL - Condense a character string

The HASHVAL instruction generates a value that is the sum of the binary values of a specified character string. You can use this value to provide a compressed form of character strings. Although other applications are possible, the following two uses are most common:

- You can use the hash value as an element in a list of nearly unique one-byte values corresponding to a list of character strings. Your program can search this list for a match condition using a computed hash value.
- You can use the hash value as an index into a table of up to 256 bytes.

Because there are far more combinations of 8-byte character strings than can be represented in one byte, duplicate hash values can result from unique character strings. Using a hash technique should provide help in dealing with this potential condition. When the number of duplicate hash values exceeds approximately one half of the total number of character strings, the hash technique begins to lose its advantage.

The algorithm used to get the hash value is as follows:

1. The character string is padded with blanks on the right to the length specified in the instruction; then, if required, the string is padded with zeros to make a total of eight characters.
2. The first four bytes are added to the second four bytes to form a partial result.
3. The first two bytes of the partial result are then added to the second two bytes, forming a second partial result.
4. The resulting two bytes are then added together forming the final result or one-byte hash total.

Syntax:

```
label         HASHVAL  'character string',RANGE=,LENGTH=,
                       TYPE=

Required:     'character string'
Defaults:     RANGE=256,LENGTH=8,TYPE=DATA
Indexable:    none
```

HASHVAL - Condense a character string *(continued)*

Operand	*Description*
character string	Code the actual character string and enclose it in quotes. The maximum length is 8 bytes (characters) unless specified as less with the LENGTH operand. If fewer characters are coded than the default or specified length, the string is padded to the right with blanks to fill the field.
RANGE=	A value from 1 to 256 that specifies the maximum range of resulting hash values (the modulus function). The resulting hash value is the remainder of the 1-byte sum divided by either the range value specified or the default value of 256.
LENGTH=	A value from 1 to 8 that specifies the maximum number of characters to be used in calculating the hash value. If you specify a character string with fewer characters than the maximum, the system pads the character string to the right with blanks until it equals the length specification.
TYPE=	EQU, assigns the resulting hash value the label you coded for the HASHVAL instruction. DATA (the default), does not equate the final hash value with the instruction label.

Syntax Examples

1) Generate a hash value of X'7F'.

```
        HASHVAL  'EIGHTCNT'
```

2) Generate a hash value of X'5C'.

```
        HASHVAL  'FOUR'
```

3) Generate a hash value of X'5A'. The value is not padded with blanks because LENGTH=4.

```
        HASHVAL  'FOUR',LENGTH=4
```

4) Generate a hash value of X'2A' (X'5C' modulus 50).

```
        HASHVAL  'FOUR',RANGE=50
```

5) Generate a hash value of X'5C' and assign the HASHVAL label this value (LABEL EQU X'5C').

```
LABEL   HASHVAL  'FOUR',TYPE=EQU
```

IDCB - Create an immediate device control block

The IDCB statement creates a standard immediate device control block that specifies a hardware operation. You must use this statement when doing EXIO processing.

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

Syntax:

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

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

Operand	*Description*
COMMAND=	The specific I/O operation. Code one of the keywords from the following list. In the following keyword list 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 for interrupts or initialization	(60)
CONTROL	- Initiate a control action to the device	(6x)
RESET	- Initiate a device reset operation	(6F)

IDCB - Create an immediate device control block *(continued)*

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 statement. See your hardware description manual to determine whether you need to code this operand for the operation you want to perform.

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 4-bit device-dependent value that modifies the command code specified by the COMMAND operand. Code one hexadecimal digit.

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

IBIT= ON (the default), to allow the device to present interrupts.

OFF, if the device should not present interrupts.

Syntax Examples

1) Transfer data to the device and set the function bit.

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

2) Prepare the device for interrupts on hardware level 3.

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

3) Start a cycle steal operation for the device.

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

IF - Test if a condition is true or false

The IF instruction determines whether a conditional statement is true or false and, based on its decision, determines the next instruction to execute.

A conditional statement can compare two data items or ask whether a bit is "on" (set to 1) or "off" (set to 0). The instruction syntax shows the general format of conditional statements used with the IF instruction.

You can compare data in two ways: *arithmetically* or *logically*. When you compare data arithmetically, the system interprets each number as a positive or negative value. The system, for example, interprets X'0FFF' as 4095. It interprets X'FFFF', however, as a -1. Though X'FFFF' seems to be a larger hexadecimal number than X'0FFF', the system recognizes the former as a negative number and the latter as a positive number. X'FFFF' is a negative number to the system because the left-most bit is "on."

When you compare data logically, the system compares the data areas byte by byte. The system interprets X'FFFF' not as a -1 but as a string of 2 bytes with all bits "on."

With EBCDIC or ASCII character data, the system makes a logical comparison of the characters byte by byte. In a logical comparison of a capital 'A' (X'C1') with a capital 'H' (X'C8'), the system recognizes the capital A to be "less than" the capital H. By comparing character data logically, you can use the IF instruction to sort items alphabetically ('a' is less than 'c' which is greater than 'b').

The syntax box shows the IF instruction with a single conditional statement. You can specify several conditional statements on a single IF instruction, however, by using the AND and OR keywords. These keywords allow you to join conditional statements. "Rules for Evaluating Statement Strings Using AND and OR" on page LR-129 provides additional information regarding use of the IF instruction. The keywords are described in the operands list and examples using the keywords are shown following the instruction description.

Syntax:

```
label      IF        (data1,condition,data2,width)

label      IF        (data1,condition,data2,width),GOTO,loc

Required:  one conditional statement
Defaults:  width is WORD for arithmetic comparison
Indexable: data1 and data2 in each statement
```

IF - Test if a condition is true or false *(continued)*

Operand	*Description*
data1	The label of a data item to be compared to data2 or the label of the data area that contains the bit to be tested.
condition	An operator that indicates the relationship or condition to be tested. The valid operators for the IF instruction are as follows:

Arithmetic and Logical Comparisons	*Testing a Bit Setting*
EQ - Equal to	ON or OFF
NE - Not equal to	
GT - Greater than	
LT - Less than	
GE - Greater than or equal to	
LE - Less than or equal to	

data2 The label of a data item to be compared to data1 or the label of the data area that contains the bit in data1 to be tested. For an arithmetic comparison, specify immediate data or the label of a data area. Immediate data can be an integer from 0 to 32767, or a hexadecimal value from 0 to 65535 (X'FFFF'). For a logical comparison, specify the label of a data area. For a bit comparison, specify immediate data.

When you check a bit setting, remember that bit 0 is the leftmost bit of the data area.

width Specify an integer number of bytes for a logical comparison (no default).

For an arithmetic comparison, you can specify one of the following:

BYTE - Bytes
WORD - Words (the default)
DWORD - Doublewords
FLOAT - Floating-points (one word, 2-byte value)
DFLOAT - Doublewords, floating-points (4-byte value)

GOTO If the statement is true and GOTO is coded, control passes to the instruction at loc. If the statement is false, execution proceeds sequentially.

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. If the condition is false, execution continues with the next ELSE statement (if one is coded) or ENDIF statement.

IF - Test if a condition is true or false *(continued)*

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 fullword boundary.

AND Enables you to join conditional statements. Code the operand between the conditional statements you want to join. The AND operand indicates that each of the conditional statements must be true before a program will execute. See the syntax examples for this instruction.

You can join several pairs of conditional statements by using several AND operands. You also can use the AND and OR operands within the same IF instruction.

OR Enables you to join conditional statements. Code the operand between the conditional statements you want to join. The OR operand indicates that one of the conditional statements must be true before a program will execute.

You can join several pairs of conditional statements by using several OR operands. You also can use the OR and AND operands within the same IF instruction.

Notes:

1. See "Rules for Evaluating Statement Strings Using AND and OR" on page LR-129 for information on use of the OR and AND operands to connect statements logically within the IF instruction.

2. Code the word THEN after the conditional statement to make the program easier to read. See Syntax Example 2.

Syntax Examples with IF, ELSE, and ENDIF

1) If A equals B, pass control to the instruction at label ERROR. This is an arithmetic comparison.

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

2) If the first 4 bytes of A are greater than or equal to the first four bytes of B, pass control to the instruction at label RETRY. This is a logical comparison.

```
IF     (A,GE,B,4),GOTO,RETRY
```

IF - Test if a condition is true or false *(continued)*

3) If C is not equal to D, execute the code that follows the IF instruction. This is an arithmetic comparison.

```
IF     (C,NE,D),THEN
   .
   .
   .
ENDIF
```

4) If register #1 is equal to 1, execute the code that follows the IF instruction; if #1 is not equal to 1, execute the code following the ELSE statement. This is an arithmetic comparison.

```
IF     (#1,EQ,1)
   .
   .
ELSE
   .
   .
ENDIF
```

5) If the first three bytes of A are less than the first three bytes of B, execute the code following the IF instruction. If the first three bytes of A are greater than or equal to the first three bytes of B, execute the code following the ELSE statement. This is a logical comparison.

```
IF     (A,LT,B,3)
   .
   .
   .
ELSE
   .
   .
   .
ENDIF
```

IF - Test if a condition is true or false *(continued)*

6) Test whether A is equal to B and whether C is equal to D. If both conditional statements are true, execute the code that follows the IF instruction; if either one or both of the conditional statements are false, execute the code following the ELSE statement. This is an arithmetic comparison.

```
IF     (A,EQ,B),AND,(C,EQ,D)
   .
   .
ELSE
   .
   .
ENDIF
```

7) If A equals B and X is greater than Y, instructions x1, x2, and x3 will execute. If A equals B, but X is not greater than Y, instructions x1 and x3 will execute. If A does not equal B, only instruction x4 executes.

```
IF   (A,EQ,B)
 x1
  IF   (X,GT,Y)
    x2
  ENDIF
 x3
ELSE
 x4
ENDIF
```

8) If the third bit starting at label A is a 1, execute the code following the IF instruction. If the third bit starting at label A is a 0, execute the code following the ELSE statement.

```
IF     (A,ON,2)
   .
   .
   .
ELSE
   .
   .
   .
ENDIF
```

IF - Test if a condition is true or false *(continued)*

9) If the bit in A at the position defined by BIT1 is a 0, execute the code following the IF instruction. If the bit is not a 0, set the value of the bit to 0.

```
IF       (A,OFF,BIT1)
  .
  .
  .
ELSE
 SETBIT  A,BIT1,OFF
ENDIF
```

IF - Test if a condition is true or false *(continued)*

Sample Conditional Statements

```
Arithmetic comparisons      Comments

(A,EQ,0)                    A equal to 0, WORD
(A,EQ,X'0022')              A equal to hexadecimal 22, WORD
(A,NE,B)                    A not equal to B, WORD
(DATA1,LT,DATA2,WORD)       DATA1 less than DATA2, WORD
(CHAR,EQ,C'A',BYTE)         CHAR equal to 'A', BYTE
(XVAL,GT,Y,DWORD)           XVAL greater than Y, DWORD
((A,#1),EQ,1)               (A,#1) equal to 1, WORD
((A1,#1),LE,(B1,#2))        (A1,#1) LE (B1,#2), WORD
(#1,EQ,1)                   #1 equal to 1, WORD
(#1,GT,#2)                  #1 greater than #2, WORD
((C,#2),EQ,CHAR,BYTE)       (C,#2) equal to CHAR, BYTE
(F1,GT,0,FLOAT)             F1 greater than 0, FLOAT
(L2,LT,L3,DFLOAT)           L2 less than L3, DOUBLEWORD FLOATING-POINT
((BUF,#1),LE,1,FLOAT)       (BUF,#1) less than or equal 1, FLOAT

D  EQU  2                   D has a word value of X'0002'
   IF   (B,EQ,+D,BYTE)      B equal to X'00' (leftmost byte of D)
```

```
Logical Comparisons         Comments

(A,EQ,B,8)                  A equal to B, 8 bytes
((BUF,#1),NE,DATA,3)        (BUF,#1) not equal to DATA, 3 bytes
(A,EQ,B,2)                  A equal to B, 2 bytes
(DATA1,LT,DATA2,3)          DATA1 less than DATA2, 3 bytes
((BUF,#1),GE,DATA,4)        (BUF,#1) greater than or equal to DATA, 4 bytes
```

```
Testing a bit               Comments

(A,ON,B)                    The bit at position B in data area A is a 1
(A,OFF,C'BB')               The bit at the hexadecimal displacement
                            represented by the characters 'BB' in data
                            area A is a 0.  Actual displacement
                            is X'C2C2'.
(DATA1,ON,X'413C')          Bit at displacement X'413C' in DATA1 is a 1.
```

Sample Conditional 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 - Provide interval timing

The INTIME instruction provides two forms of interval timing information, reltime and loc. The first form, reltime, is a 2-word area in your program where INTIME stores a value each time an INTIME instruction executes. This value is equal to the elapsed time since system IPL. The count is expressed in milliseconds and is in double-precision integer format. The maximum value for reltime is reached after approximately 49 days of continuous operation. The system resets the counter to 0 at that time.

The second form, loc, is a single-precision integer variable where INTIME stores 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 65,535 milliseconds (65.535 seconds).

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

Syntax:

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

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

Operand	*Description*
reltime	The label 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	The label of a buffer of data area 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 "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

INTIME - Provide interval timing *(continued)*

Coding Example

When the INTIME instruction executes, it places the number of milliseconds that have elapsed since system IPL in the UPTIME variable. Because the LOC variable refers to a BUFFER statement and automatic indexing is used, the interval count since execution of the previous INTIME instruction will be placed in the next available BUFFER location. The PRINTEXT and PRINTNUM instructions print the data on the appropriate forms.

```
GETTIME  EQU      *
         INTIME   UPTIME,INTERVAL,INDEX      GET TIME IN MILLISECONDS
         DIVIDE   UPTIME,1000,DWORD          CONVERT TIME TO SECONDS
         DIVIDE   UPTIME,3600,DWORD          DIVIDE TO GET HOURS
         DIVIDE   TASK,60,RESULT=MIN         DIVIDE THE REMAINDER TO
*                                            GET MINUTES
         ENQT     $SYSPRTR
         PRINTEXT '@ADDITIONAL 100 BARRELS OF OIL                    X
               PROCESSED AT HR:MIN'
         PRINTNUM UPTIME,TYPE=D
         PRINTNUM MIN
         PRINTEXT '@AFTER BEGINNING OF PROCESSING RUN@'
         PRINTEXT '@CURRENT BATCH TOOK '
         MULT     ENTRIES,2,RESULT=INDX
         MOVEA    #1,INTERVAL
         ADD      #1,INDX
         DIVIDE   (0,#1),1000,RESULT=SECONDS
         PRINTNUM SECONDS
         PRINTEXT ' SECONDS TO PRODUCE@'
         DEQT
           .
           .
           .
UPTIME   DATA     2F'0'
MIN      DATA     F'0'
SECONDS  DATA     F'0'
INTERVAL BUFFER   1000,WORDS,INDEX=ENTRIES
INDX     DATA     F'0'
```

IOCB - Define terminal characteristics

The IOCB statement defines a terminal name and terminal characteristics for use with the ENQT instruction. You can use this statement to temporarily change such terminal characteristics as screen or page margins. You define these and other terminal characteristics during system generation. When your program releases control of a terminal, the characteristics you defined with the IOCB statement are no longer in effect.

Do not code PAGSIZE, TOPM, BOTM, LEFTM, RIGHTM, or NHIST IOCB instruction operands for a 3101 in block mode:

When coding the IOCB instruction, you can include a comment which will appear with the instruction on your compiler listing. If you include a comment, you must specify at least one operand with the instruction. The comment must be separated from the operand field by one or more blanks and it may not contain commas.

Syntax:

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

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

Operand	*Description*
name	The name of a terminal as defined by the label on a TERMINAL definition statement used in system generation. See the *Installation and System Generation Guide* for a description of the TERMINAL definition statement. This operand generates an 8-character EBCDIC string, padded as necessary with blanks, whose label is the label on the IOCB instruction. It may, therefore, be modified by the program. If unspecified, the string is blank and implicitly refers to the terminal which is currently in use by the program.

IOCB - Define terminal characteristics *(continued)*

Note: Except for the BUFFER operand, the following operands have default values established by the TERMINAL definition statement

PAGSIZE= The physical page size (form length) of the I/O medium. Specify an integer between 1 and the maximum value which is meaningful for the device. For printers, specify the number of lines per page. For screen devices, specify the size of the screen in lines. This operand is not required for the 4978, 4979, or 4980 display terminal.

If you specify this operand, BOTM must be between TOPM plus NHIST, AND PAGSIZE-1. Otherwise, unpredictable results will occur.

TOPM= The top margin (a decimal number between zero and PAGSIZE-1) to indicate the top of the logical page within the physical page for the device. The default is 0.

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

LEFTM= The left margin, the character position at which input or output begins. The default is 0. Specify a decimal value between zero and LINSIZE-1.

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

If a BUFFER statement is not specified, the default is LINSIZE-1. If a BUFFER statement is specified, the value you specify should be one less than the buffer size value.

SCREEN= ROLL, the default, for screens that are to be operated similar to a typewriter. For screen devices which are attached through the teletypewriter adapter, ROLL indicates that the system will pause when a screen-full condition occurs during continuous output.

STATIC, for a full-screen mode of operation, if full-screen mode is supported for the device. For the 3101 Display Terminal, STATIC is valid only for block mode.

NHIST= The number of history lines to be retained when a page eject is done on the 4978, 4979 or 4980 display. The default is 0. The line at TOPM+NHIST corresponds to logical line zero for the terminal I/O instructions. When a page eject (LINE=0) is performed, the screen area from TOPM to TOPM+NHIST-1 will contain lines from the previous page.

IOCB - Define terminal characteristics *(continued)*

OVFLINE= YES, if output lines which exceed the right margin are to be continued on the next line.

NO, the default, if the lines are not to be continued.

The overflow condition occurs when the system buffer (or a buffer in an application program) becomes full and the application program has taken no action to write the buffer to the device.

BUFFER= If the application requires a temporary I/O buffer of a different size from 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. The buffer size then temporarily replaces the LINSIZE value and is also the maximum amount that can be read or written at a time. For data entry applications which require full screen data transfers, for example, this avoids 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, and 4980 displays, 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 buffer size is less than the 80-byte line size of the 4978, 4979, or 4980 display, the logical screen boundaries are adjusted accordingly. If the RIGHTM is not specified or has a value greater than the buffer size, it is adjusted to one less than the buffer size value. Portions of the screen outside this range are not accessible by the application program.

Direct I/O Considerations

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. In this case the data is transferred immediately and the new line character (for carriage return, line feed, and so on) is not recognized.

When doing direct output operations, you must insert the output character count in the index word of the BUFFER before 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, 4980, and 3101 displays or buffered teletypewriter terminals. On execution of DEQT, the buffer defined by the TERMINAL statement is restored.

IOCB - Define terminal characteristics *(continued)*

Coding Example

The following example shows a use of the IOCB instruction.

In this program an ENQT instruction enqueues an IOCB whose label is TERMINAL. The IOCB instruction refers to a terminal that was assigned the label TERM24 during system generation. If no terminal named TERM24 had been defined in the system generation, the terminal currently in use by the program would be used by default. The IOCB defines a logical static screen that is 40 columns wide and 12 rows deep, in the middle of the physical display.

The terminal does not use the system-defined buffer for I/O operations, but instead uses a program-defined data buffer area called BUFR. The terminal retains the characteristics defined in the IOCB until the program executes a DEQT or PROGSTOP instruction.

```
          .
          .
          .
GETPRTR   EQU     *
          ENQT    TERMINAL
          .
          .
TERMINAL  IOCB    TERM24,TOPM=6,BOTM=17,LEFTM=20,RIGHTM=59,              C
                 SCREEN=STATIC,BUFFER=BUFR
BUFR      BUFFER 480,BYTES
          .
          .
```

IODEF - Assign a symbolic name to a sensor-based I/O device

The I/O definition statement (IODEF) defines the hardware address and attributes of a sensor-based I/O device and assigns a label to that device.

The device label consists of two characters that define the type of sensor-based I/O device you are using, followed by a number from one to 99 that identifies the individual device. The types of devices are: AI (Analog Input), AO (Analog Output), DI (Digital Input), DO (Digital Output), and PI (Process Interrupt).

You use the label assigned by IODEF to code a sensor-based I/O instruction (SBIO). The SBIO instruction only refers to the label of the I/O device. You specify the actual physical address of the device and the device attributes on the IODEF statement. (See the SBIO instruction for more details on using the symbolic device name.) The WAIT and POST instructions refer to the IODEF Process Interrupt statement.

Each IODEF statement creates an SBIO control block (SBIOCB). The control block provides the link between the IODEF statement and the SBIO instruction that refers to it. The control block also provides a location into which your program can read data or from which it can write data. The system stores data in the control block if you have not specified another storage location on the SBIO instruction. The contents of the SBIOCB are described in the *Internal Design*.

Each type of sensor-based I/O device requires a specific type of IODEF statement. You must group all IODEF statements that refer to the same type of device together in your application program. In addition, you must place all IODEF statements in your program before the SBIO instructions that refer to them.

In EDL, All IODEF statements must be in the same assembly module as the TASK or ENDPROG statement. If the SBIO instructions are to be in a separate module, you can provide symbolic names using ENTRY/EXTRN statements. You must create a separate IODEF for each task; different tasks cannot use the same IODEF statement.

The syntax of the IODEF statement for each device type (AI, AO, DI, DO, and PI) appears on the following pages.

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

IODEF (Analog Input)

Syntax:

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

Required:      AIx,ADDRESS=,POINT=
Defaults:      RANGE=5V, ZCOR=NO
Indexable:     none
```

Operand	*Description*
AIx	Analog Input, where "x" is the number (1–99) you assign to an I/O device to identify it in your application program. If you include more than one IODEF AIx statement in the program, you must group these statements together.
ADDRESS=	A two-digit hexadecimal address.
POINT=	The analog input point. The point is 0 – 7 for AI relay or 0 – 15 for AI solid state.
RANGE=	Range for the multirange amplifier. 5V = 5 Volts 500MV = 500 Millivolts 200MV = 200 Millivolts 100MV = 100 Millivolts 50MV = 50 Millivolts 20MV = 20 Millivolts 10MV = 10 Millivolts
ZCOR=	YES, to use the zero-correction facility of AI. NO (the default), not to use the zero-correction facility.

Syntax Example

Define an analog input device with the label AI1.

```
INPUT    IODEF  AI1,ADDRESS=72,POINT=1,RANGE=50MV,ZCOR=YES
```

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

IODEF (Analog Output)

Syntax:

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

Required:      AOx,ADDRESS=
Defaults:      POINT=0
Indexable:     none
```

Operand	***Description***
AOx	Analog Output, where "x" is the number (1–99) you assign to an I/O device to identify it in your application program. If you include more than one IODEF AOx statement in the program, you must group these statements together.
ADDRESS=	A two-digit hexadecimal address.
POINT=	The analog output point. The point range is 0 – 1.

Syntax Example

Define an analog output device with the label AO2.

```
OUTPUT     IODEF   AO2,ADDRESS=75,POINT=1
```

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

IODEF (Digital Input)

Syntax:

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

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

Operand	***Description***
DIx	Digital input, where "x" is the number (1–99) you assign to an I/O device to identify it in your application program. If you include more than one IODEF DIx statement in the program, you must group these statements together.
TYPE=	The type of DI operation you are performing. Code one of the following:

GROUP	The I/O operations will use the entire group of 16 DI points. DI operates in unlatched mode.
SUBGROUP	The I/O operations will use a subset of the 16-bit group. The subgroup is stored right-adjusted in the input word with the leftmost bits set to zero. DI operates in unlatched mode.
EXTSYNC	The I/O operations will use the hardware external synchronization feature for DI. You must code the count field on the associated SBIO instructions. DI operates in latched mode.

ADDRESS=	A two-digit hexadecimal address.
BITS=(u,v)	The portion of the 16-point group you are using when you specify TYPE=SUBGROUP. The portion starts at bit u (0 to 15) for a length of v (1 to 16-u).

Syntax Example

Define a digital input device with the label DI1.

```
INPUT    IODEF  DI1,TYPE=GROUP,ADDRESS=49
```

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

IODEF (Digital Output)

Syntax:

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

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

Operand	*Description*
DOx	Digital output, where "x" is the number (1–99) you assign to an I/O device to identify it in your application program. If you include more than one IODEF DOx statement in the program, you must group these statements together.
TYPE=	The type of DO operation you are performing. Code one of the following:

GROUP	The I/O operations will use the entire group of 16 DO points.
SUBGROUP	The I/O operations will use a subset of the 16-bit group. Bits that are not part of the subset you specify remain unchanged.
EXTSYNC	The I/O operations will use the hardware external synchronization feature for DO. You must code the count field on the associated SBIO instructions.

ADDRESS=	A two-digit hexadecimal address.
BITS=(u,v)	The portion of the 16-point group you are using when you specify TYPE=SUBGROUP. The portion starts at bit u (0 to 15) for a length of v (1 to 16-u).

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

Syntax Examples

1) Define a digital output device with the label DO1. The I/O operations will use the entire group of 16 DO points (TYPE=GROUP).

```
OUTPUT     IODEF  DO1,TYPE=GROUP,ADDRESS=4B
```

2) Define a digital output device with the label DO2. The I/O operations will use the hardware external synchronization feature (TYPE=EXTSYNC).

```
OUTPUT2    IODEF  DO2,TYPE=EXTSYNC,ADDRESS=4A
```

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

IODEF (Process Interrupt)

Syntax:

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

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

Operand ***Description***

PIx Process interrupt, where "x" is the number (1–99) you assign to an I/O device to identify it in your application program. If you include more than one IODEF PIx statement in the program, you must group these statements together.

ADDRESS= A two-digit hexadecimal address.

BIT= The bit number (0 – 15) used for PI.

TYPE= Indicates when the system will invoke the special process interrupt routine you provide. Code one of the following:

GROUP The supervisor gives control to the special interrupt routine you provide if an interrupt occurs on any bit in the PI group. The PI group is not read or reset; reading or resetting the PI group is the responsibility of your routine.

Control returns to the supervisor with a branch to the entry point SUPEXIT. You must include the module $EDXATSR with your program to use SUPEXIT. If the routine processes the interrupt on level 0, it can issue a Series/1 hardware exit level instruction (LEX) instead of returning to SUPEXIT. Issuing the LEX instruction greatly improves performance.

Note: To use TYPE=GROUP, you must be familiar with the operation of the Series/1 process interrupt feature. Your routine must contain all the instructions necessary to read and reset the process interrupt group to which it refers.

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

BIT — The supervisor gives control to the special interrupt routine you provide only when an interrupt occurs on the bit specified in the BIT= operand.

When control returns to the supervisor, the contents of R1 must be the same as when the system invoked your routine and R0 must contain either '0' or a POST code. If R0 contains a POST code, R3 must contain the address of an ECB to be posted by the POST instruction.

Register 7 contains the supervisor return address on entry. If your routine is in partition 1, you can return control to the supervisor by using the assembler instruction **BXS (R7)**. The SPECPIRT instruction allows you to return control to the supervisor from any partition. (See the SPECPIRT instruction for a coding description.)

SPECPI= The label of the first instruction of a special process interrupt routine. You must write the routine in Series/1 assembler language.

The supervisor executes the routine when the defined interrupt occurs. This routine bypasses the normal supervisor response and allows you to handle process interrupts quickly.

You can include more than one special process interrupt routine in your program.

Syntax Examples

1) Define a process interrupt device with the label PI1.

```
A       IODEF   PI1,ADDRESS=48,BIT=2
```

2) Define a process interrupt device with the label PI2.

```
B       IODEF   PI2,ADDRESS=49,BIT=15
```

IODEF - Assign a symbolic name to a sensor-based I/O device *(continued)*

Coding Examples

1) The supervisor passes control to the special interrupt routine FASTPI1 when an interrupt occurs on bit 3.

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

FASTPI1 EQU      *

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

2) The supervisor passes control to the special interrupt routine labeled FASTPI2 when an interrupt occurs on any one of the PI group bits at address 49.

```
        IODEF PI6,ADDRESS=49,TYPE=GROUP,SPECPI=FASTPI2
          .
FASTPI2 EQU       *
```

IOR - Compare the binary values of two data strings

The Inclusive OR instruction (IOR) compares the binary value of operand 2 with the binary value of operand 1. The instruction compares each bit position in operand 2 with the corresponding bit position in operand 1 and yields a result, bit by bit, of 1 or 0. If either or both of the bits compared is 1, the result is 1. If neither of the bits compared is 1, the result is 0.

Syntax:

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

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

Operand	*Description*
opnd1	The label of the data area to be compared with opnd2. Opnd1 cannot be a self-defining term.
opnd2	The value to be compared with opnd1. You can specify a self-defining term or the label of a data area.
count	Specify the number of consecutive values in opnd1 on which the operation is to be performed. The maximum value allowed is 32767.

The count operand can include the precision of the data. Select one precision which the system uses for opnd1, opnd2, and the resulting bit string. When specifying a precision, code the count operand in the form,

```
(n,precision)
```

where "n" is the count and "precision" is one of the following:

BYTE -- byte precision
WORD -- word precision (default)
DWORD -- doubleword precision

The precision you specify for the count operand is the portion of opnd2 that is used in the operation. If the count is (3,BYTE), the system compares the first byte of data in opnd2 with the first three bytes of data in opnd1.

RESULT= The label of the data area or vector in which the result is to be placed. When you specify RESULT, the value of opnd1 does not change during the operation. This operand is optional.

IOR - Compare the binary values of two data strings *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) Compare X'F008' with the contents of STRING and place the result in the data area labeled ANS.

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

After the IOR operation, ANS contains:

Hexadecimal -- X'FF08'

Binary -- 1111 1111 0000 1000

2) Compare the contents of OPER2 to the first three doublewords beginning at label OPER1 and place the result in the data area labeled RESULTX.

```
          IOR     OPER1,OPER2,(3,DWORD),RESULT=RESULTX
           .
           .
           .
OPER1     DC      X'FFFF'     binary 1111 1111 1111 1111
          DC      X'0000'     binary zeros
          DC      X'8888'     binary 1000 1000 1000 1000
          DC      X'4567'     binary 0100 1010 0110 0111
          DC      X'1111'     binary 0001 0001 0001 0001
          DC      X'AAAA'     binary 1010 1010 1010 1010
OPER2     DC      2X'AAAA'
RESULTX   DC      6F'0'
```

After the operation, RESULTX contains:

Hexadecimal -- X'FFFF AAAA AAAA EAEF BBBB AAAA'

IOR - Compare the binary values of two data strings *(continued)*

3) Compare the first byte of data in TEST to the first three bytes of data in INPUT. Place the result in the data area labeled OUTPUT.

```
           IOR    INPUT,TEST,(3,BYTE),RESULT=OUTPUT
            .
            .
            .
INPUT      DC     C'1.2'   binary  1111 0001 0100 1010 1111 0010
TEST       DC     C'0.0'   binary  1111 0000
OUTPUT     DC     3C'0'    binary  1111 0000 1111 0000 1111 0000
            .
```

After the operation, OUTPUT contains:

Binary -- 1111 0001 1111 1010 1111 0010

LASTQ - Acquire the last queue entry in a chain

The LASTQ instruction acquires the last (most recent) entry in a queue. You define a queue with the DEFINEQ statement. The queue entry can contain data or the address of a data buffer. After you acquire the contents of the queue entry, the system adds the entry to the free chain of the queue.

Syntax:

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

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

Operand	*Description*
qname	The name of the queue from which the entry is to be fetched. The queue name is the label on the DEFINEQ statement that creates the queue.
loc	The label of a word of storage where the entry is placed. #1 or #2 can be used.
EMPTY=	Specify the first instruction of the routine to be invoked if a "queue empty" condition is detected during the execution of this instruction. If this operand is not specified, control returns to the next instruction after the LASTQ. A return code of -1 in the first word of the task control block indicates that the operation completed successfully. A return code of +1 indicates that the queue is empty.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

See the examples following the NEXTQ instructions.

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
1	Queue is empty

LOAD - Load a Program

The LOAD instruction allows one program to load another main program or overlay program from a program library on disk or diskette. The loaded program runs parallel with, and independently of, the loading program, regardless of whether it is a main program or an overlay. The loading program may, however, synchronize its own execution with the loaded program.

The LOAD instruction also allows you to load a program in another partition and to pass that program parameters. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example of such a cross-partition operation. Refer to the *Event Driven Executive Language Programming Guide* for more information on cross-partition services.

A program can 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 is brought into storage more quickly than by an independent LOAD

Figure 9 on page LR-267 illustrates the two ways of loading a program.

You can test the first word of the task control block (TCB) of the loading program to determine the result of the load operation. The label of the TCB is the label of the program (taskname). If this word is -1, the operation was successful.

When a LOAD instruction loads 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.

LOAD - Load a Program *(continued)*

Syntax:

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

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

Operand	*Description*
progname	The 1-8 character name of a program stored in an Event Driven Executive library. You can specify the volume from which to load the program by separating the program name and the volume name by a comma and enclosing both in parentheses. To load program PROGA on volume EDX003, you would code: (PROGA,EDX003). The program must reside on disk or diskette. The volume name can be 1-6 characters long.
PGMx	The parameter "x" is a number from 1 to 9 that specifies 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 label of the first word in a list of consecutive parameter words to be passed to the loaded program. (See the PROGRAM statement for the maximum length of this list.)
DEQT=	YES (the default), dequeues the terminal currently in use by the loading program. NO, prevents the terminal from being dequeued when the LOAD instruction executes. Coding DEQT=NO also forces the LOGMSG operand to LOGMSG=NO. **Note:** Allow this operand to default or code DEQT=YES for a virtual terminal program.
DS=	The names of the data sets to be passed to the loaded program. If your program loads another program, you can pass the loaded program the names of 1 to 9 data sets. This operand enables the main program to define,

LOAD - Load a Program *(continued)*

during the load operation, the names of the data sets the loaded program will use. On the PROGRAM statement of the program to be loaded, the data set list contains the sequence '??' for each missing data set name. This sequence indicates that the data set name will be supplied at load time. (See the PROGRAM statement for more information.)

For example, if the PROGRAM statement in the program to be loaded contained the data set list:

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

the LOAD instruction in the main program,

```
LOAD   MYPROG,DS=(MYDATA)
```

would pass the data set name MYDATA to the loaded program and produce the following list for the loaded program:

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

The LOAD instruction, in this case, replaces the sequence '??' with the data set name MYDATA.

When the main program loads an overlay program, you must code DSx, where "x" is the relative position (number) of the data set in the list of data set names on the PROGRAM statement of the main program.

The parameter "x" can be a number from 1 to 9. For example, to pass the second data set name in a list to an overlay program named OVPGM, you would code:

```
LOAD    OVPGM,DS=DS2
```

All unspecified data set names in the program being loaded must be resolved at LOAD time or the load operation will fail.

If the main program passes a tape data set to another program, the main program's data set control block (DSCB) is no longer associated with the tape data set. This allows the loaded program to have access to the tape data set using the main program's DSCB. When the loaded program ends, the system closes the tape data. If the main program needs to use the tape data set again, the main program must call DSOPEN or load $DISKUT3 to reopen the tape data set.

LOGMSG= YES (the default), to print or display the "PROGRAM LOADED" message on the terminal being used by the program.

LOAD - Load a Program *(continued)*

NO, to avoid printing or displaying this message.

EVENT= The label of an event (ECB statement) that the system posts complete when the loaded program issues a PROGSTOP.

By issuing a LOAD and a subsequent WAIT for this event, the main program can synchronize its own execution with the loaded program. The ECB, however, must not be reset with a RESET instruction or with the RESET operand of a WAIT instruction, or synchronization may be lost.

Note: If you specify this operand, the main program must wait for the loaded program to end. Otherwise, the system will post the ECB when the loaded program ends even though the main program may no longer be active. The results, in such a case, are unpredictable.

PART= The number of the partition in which you want to load the program. The system loads the program in the same partition the main program resides in if you do not code this operand. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example of loading a program in another partition.

You can code one of the following:

- A number from 1 to 8 (partition 1 to 8).
- PART=ANY, to load the program in any available partition.
- The label of a 1-word data area that contains the partition number.

If the data area contains a zero, the system loads the program in any available partition.

Do not use this operand if the main program loads an overlay program.

ERROR= The label of the first instruction of the routine to receive control if an error condition occurs during the load operation. If you do not code this operand, control passes to the instruction following the LOAD instruction and you can test for errors by referring to the return code in the first word of the task control block (TCB).

STORAGE= The number of bytes of additional storage to be added to the loaded program. This operand overrides the value of the STORAGE operand on the PROGRAM statement of the program to be loaded.

Some application programs have a minimum storage requirement; be sure you know what it is before using this override. The load operation will fail if the loaded program requires more storage than is available. (See the PROGRAM statement for more information on allocating program storage.)

LOAD - Load a Program *(continued)*

This operand does not override the STORAGE operand on the PROGRAM statement if you code a 0 or allow the operand to default.

Do not use this operand if the main program loads an overlay program.

P2= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

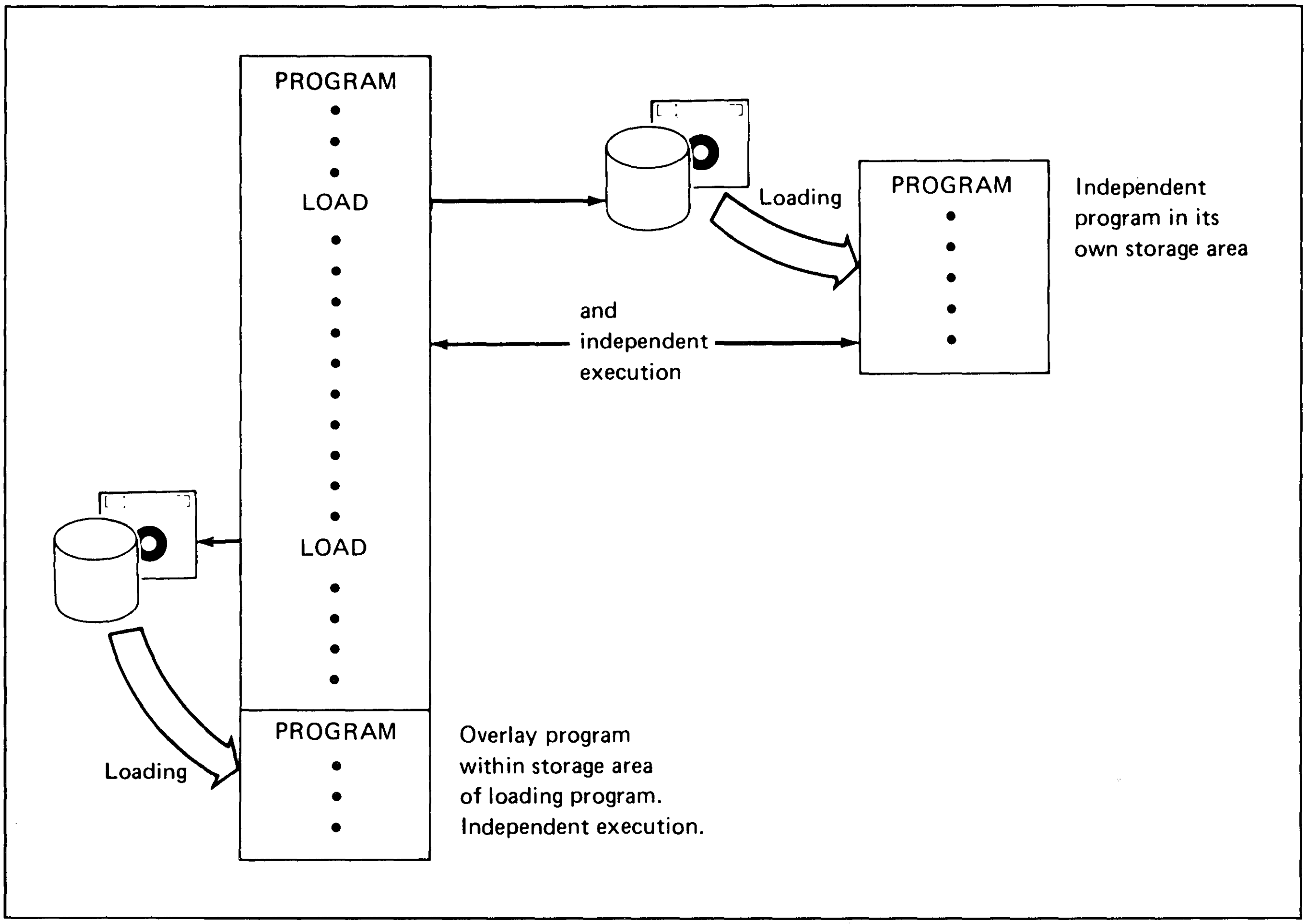

Figure 9. Two Ways of Loading a Program

LOAD - Load a Program *(continued)*

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program issuing the instruction.

Return Code	Condition
-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(ette) 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	One of the data sets 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 (two or more programs referenced the same volume).
81	Cross partition LOAD requested, support not included at system generation.
82	Requested partition number greater than number of partitions in the system.
83	Load instruction attempted to access a 1024 bytes/sector diskette without $IO1024 pre-loaded in storage.

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.

MECB - Create a list of events

The MECB statement creates a control block for use by a WAITM instruction. The control block contains control information and a list of the ECBs for the events on which the WAITM instruction must wait.

You can specify labels for several of the fields in the MECB so that you can get access to them from your application program. The fields you can get access to are:

- The number of events posted
- The pointer to the last (most recent) event posted
- The post code received by each event in the list.

You must use the ECB statement to code the necessary ECBs in programs assembled under $EDXASM, except for those ECBs specified with the EVENT= operand on the LOAD instruction or on the PROGRAM or TASK statement. In programs assembled with the host or Series/1 macro assemblers, the system automatically generates an ECB for an event named in a POST instruction.

See "WAITM - Wait for one or more events in a list" on page LR-523 for the description and syntax of the WAITM instruction. See "ECB - Create an event control block" on page LR-136 for the description and syntax of the ECB statement.

Note: To use the MECB statement, you must have included the SWAITM module in your system and specified the MECBLST keyword on the system statement during system generation. (Seethe *Installation and System Generation Guide* for additional information.)

Syntax:

```
label      MECB      (ecb1,ecb2,...ecbn),nwait,MAXECB=,
                     CODE=,NUMP=,LAST=,P1=(lbl1,lbl2,...lbln),P2=

Required:   label
Defaults:   nwait=1, CODE=-1,
            MAXECB=number of ECB labels coded
Indexable:  none
```

Operand	*Description*
ecb1,ecb2, ...,ecbn	The label of each ECB you are including in the MECB list. The system generates additional blank entries if the number of labels is less than the value coded for MAXECB=.
nwait	The number of events that must occur before the waiting program can continue.
MAXECB=	The number of events (ECBs) in the MECB list. If this value is larger than the number of ECB labels coded, the system generates blank entries to make up the difference.

MECB - Create a list of events *(continued)*

CODE= The initial value of the MECB post code. If this word is not zero when your program issues the WAITM instruction, the system does not wait unless the WAITM instruction has the RESET operant coded. (The default is -1.)

NUMP= The label for the field containing the number of events posted.

LAST= The label for the pointer to the last event posted.

P1=(...) Parameter naming operand. Specify labels for the fields in the MECB that contain the post code for the respective ECBs. (The system places the post code received by an ECB in the first word of the MECB entry for that ECB.)

P2= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Wait for two of the three specified events to occur before continuing. Place labels on the pointer to the last event that occurred and on the post codes.

```
MECB1    MECB    (ECB1,ECB2,ECB3),2,LAST=LASTECB,P1=(POST1,POST2,POST3)
```

MESSAGE - Retrieve a program message

The MESSAGE instruction retrieves a formatted program message from a data set or module and displays or prints the message. See Appendix E, "Creating, Storing, and Retrieving Program Messages" on page LR-615 for more information.

The instruction also allows you to include data or text generated by your program within the message.

Syntax:

```
 label        MESSAGE msgno,COMP=,SKIP=,LINE=,SPACES=,
                      PARMS=(parm1,...,parm8),MSGID=,
                      XLATE=,PROTECT=,P1=

Required:    msgno,COMP=
Defaults:    MSGID=NO,XLATE=YES,PROTECT=NO
Indexable:   none
```

Operand	*Description*
msgno	The number of the message you want displayed or printed. This operand must be a positive integer or a label preceded by a plus sign (+) and equated to a positive integer.
COMP=	The label of the COMP statement that points to the data set or module that contains the formatted program messages. See the COMP statement description for more information.
SKIP=	The number of lines to be skipped before the system prints or displays the message. If your cursor is at line 2 on a display screen and you coded SKIP=6, the system displays the message on line 8. For a printer, the SKIP operand controls forms movement. The SKIP operand causes the system to display or print the contents of the system buffer. If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

MESSAGE - Retrieve a program message *(continued)*

LINE= The line number on which the message is to be printed or displayed. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system prints or displays the message at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system displays the message on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to print the message. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the message appears on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system prints or displays the message. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

PARMS= The labels of data areas containing information to be included in the message. You can code up to eight labels. If you code more than one label, you must enclose the list in parentheses.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

MESSAGE - Retrieve a program message *(continued)*

MSGID= YES, if you want the message number and four-character prefix to be printed at the beginning of the message you are retrieving from a data set or module containing formatted program messages. See the COMP statement operand 'idxx' for a description of the four-character prefix.

NO (the default), to avoid printing this information.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

XLATE= NO, to send the message to the terminal as is, without translation. Code this option if the message contains special characters that should not be altered or interpreted by the terminal.

YES (the default), to cause translation of characters from EBCDIC to the code the terminal uses to display the message.

With a 3101 in block mode, XLATE=NO also prevents the system from inserting the attribute byte and escape sequences into the message, and overrides the effects of TERMCTRL SET,STREAM=YES.

Note: For a description of 3101 escape sequences refer to *IBM 3101 Display Terminal Description*, GA18-2033.

PROTECT= YES, to write protected characters to a static screen device that supports this feature, such as an IBM 4978, 4979, 4980, or 3101 in block mode. Protected characters cannot be typed over.

NO (the default), to avoid writing protected characters to a static screen.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

MESSAGE - Retrieve a program message *(continued)*

Syntax Examples

1) Retrieve and print the first message in the disk data set to which the COMP statement points.

```
MSG1      MESSAGE    1,COMP=MSGSET
             .
             .
             .
          PROGSTOP
MSGSET    COMP       'ERRS',DS1,TYPE=DSK
```

2) Retrieve and print the fifth message in the module to which the COMP statement points. Insert the parameter "ACCOUNTS" in the message.

```
MSG2      MESSAGE    +MSG,PARMS=A,COMP=MSGSET
             .
             .
          PROGSTOP
MSG       EQU        5
A         DATA       C'ACCOUNTS'
MSGSET    COMP       'ERRS',ERRORS,TYPE=STG
```

Coding Example

The following example uses the MESSAGE instruction to retrieve and print a message contained in a disk data set. The program TASK loads a second program called CALCPGRM. A WAIT instruction suspends the execution of TASK until CALCPGRM completes. When CALCPGRM finishes, it posts the ECB at label LOADECB. The MESSAGE instruction at label MSG1 retrieves the first message in the disk data set MSGDS1 on volume EDX002. The first message in this data set is:

<<PROGRAM>> HAS FINISHED PROCESSING/*

MESSAGE - Retrieve a program message *(continued)*

The MESSAGE instruction inserts the parameter CALCPRGM into the "PROGRAM" field of the message and displays the message as follows:

STAT0001 CALCPGRM HAS FINISHED PROCESSING

Because the MESSAGE instruction contains MSGID=YES, the number of the message and the four-character prefix "STAT" appear at the beginning of the message. The COMP statement assigns the four-character prefix to the message.

```
TASK      PROGRAM    START,DS=((MSGDS1,EDX002))
LOADECB   ECB
START     EQU         *
            .
            .
          LOAD       CALCPGRM,EVENT=LOADECB
          WAIT       LOADECB
MSG1      MESSAGE    1,COMP=MSGSET,SKIP=1,PARMS=A,MSGID=YES
            .
            .
          PROGSTOP
A         DATA       'CALCPGRM'
MSGSET    COMP       'STAT',DS1,TYPE=DSK
          ENDPROG
          END
```

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
301-316	Error while reading message from disk. Subtract 300 from this value to get the actual return code. See the disk return codes following the READ or WRITE instruction for a description of the code.
326	Message number out of range
327	Message parameter not found
328	Instruction does not supply message parameter(s)
329	Invalid parameter position
330	Invalid type of parameter
331	Invalid disk message data set
332	Disk message read error
333	Storage-resident module not found
334	Message parameter output error
335	Disk messages not supported (MINMSG support only)

MOVE - Move data

The MOVE instruction moves data from operand 2 to operand 1. If operand 2 is "immediate data," it must meet the requirements listed in the opnd2 description.

For an example of moving data across partitions, see Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559. Refer to the *Event Driven Executive Language Programming Guide* for more information on cross-partition services.

Syntax:

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

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

Operand	*Description*
opnd1	The label of the data area to receive the data from opnd2. Opnd1 cannot be a self-defining term.
opnd2	The value moved into opnd1. You can specify a self-defining term or the label of a data area.

If opnd2 is a self-defining term, it must be one of the following:

- An integer, whose value is between -32768 and +32767
- An EBCDIC character string of one or two bytes, enclosed in single quotes, and preceded by the constant type indicator C
- A hexadecimal character string of 1 - 4 hexadecimal digits, enclosed in single quotes, and preceded by the constant type indicator X

count The number of consecutive values on which the operation is to be performed. Do not code a label for count. The maximum value allowed for the count operand is 32767.

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

BYTE -- Byte precision
WORD -- Word precision (the default)
DWORD -- Doubleword precision

MOVE - Move data *(continued)*

FLOAT -- Single-precision floating-point
DFLOAT -- Extended-precision floating-point

You can substitute the precision specification for the count specification, in which case the count defaults to 1, or the precision specification can accompany the count in the form of a sublist: (count,precision). For example, MOVE A,B,BYTE is equivalent to MOVE A,B,(1,BYTE). When using the sublist form of the count operand, you must specify both the count and the precision.

For all precisions other than BYTE, opnd1 and opnd2 must specify even addresses.

The precision is always BYTE when you do a cross-partition MOVE operation. For example, MOVE A,B,(4,DWORD) becomes MOVE A,B,(16,BYTE). This precision change is important to remember when you use the P3= operand to change the count. The instruction,

```
MOVE  A,B,(4,WORD),FKEY=0,P3=COUNT
```

really has a count of 8 bytes. If you want to change the count to (2,WORD), you must move a value of 4 into COUNT.

If FLOAT or DFLOAT precision is specified, the system converts the immediate data field to floating-point format.

If BYTE precision is specified and opnd2 is immediate data, the system moves different bytes of opnd2 depending on which assembler is used. The macro assembler causes the system to move the rightmost byte of opnd2, while $EDXASM causes the system to move the leftmost byte of opnd2.

For example, if the following is coded:

```
Q    EQU    X'1234'
     MOVE   HERE,+Q,(1,BYTE)
```

The system moves X '34' to location HERE if the instruction is assembled with a macro assembler. The system moves X '12' to location HERE if the instruction is assembled with $EDXASM.

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 in the (parameter,#r) format. See "Software Register Usage" on page LR-10 for further information.

MOVE - Move data *(continued)*

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). See "Software Register Usage" on page LR-10 for further information.

If you specify TKEY and opnd2 is immediate data, opnd2 is always one word in length regardless of the precision specified. The values you code for the precision and the count operand determine the amount of data that is moved.

When you specify byte precision in a cross-partition move and opnd2 is immediate data, the system reads an entire word of data and moves that word one byte at a time. For example, if opnd2 is X'F5', the system reads that value as X'00F5' and moves X'00' as the first byte.

Px= Parameter naming operands. If P3 is coded, only the count operand is altered. The precision specification remains unchanged. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to use these operands.

MOVE - Move data *(continued)*

Syntax Examples

The following syntax examples show the variety of ways you can code the MOVE instruction:

1) Move a word of B to A.

```
MOVE    A,B
```

2) Move 64 EBCDIC blanks to TEXT.

```
MOVE    TEXT,C' ',(64,BYTE)
```

3) Move 16 words of V2 to V1.

```
MOVE    V1,V2,16
```

4) Move the contents of index register 1 to SAVE.

```
MOVE    SAVE,#1
```

5) Move contents of INDEX into index register 2.

```
MOVE    #2,INDEX
```

6) Move four doublewords of C to D.

```
MOVE    D,C,(4,DWORD)
```

7) Move a single-precision floating-point value from F1 to F2.

```
MOVE    F2,F1,(1,FLOAT)
```

8) Move the address of $START into index register 1.

```
MOVE    #1,+$START
```

9) Move six doubleword floating-point numbers (24 words) from L1 to LR.

```
MOVE    LR,L1,(6,DFLOAT)
```

10) Move ten floating-point zero values to the indexed address of (BUF,#1).

```
MOVE    (BUF,#1),0,(10,FLOAT)
```

MOVE - Move data *(continued)*

11) Move one word from $START in partition 1 to HERE.

```
MOVE    HERE,$START,FKEY=0
```

12) Move the contents of index register 2 to the indexed address (0,#1) in a partition defined by KEY.

```
MOVE    (0,#1),#2,TKEY=KEY
```

13) Move four words of blanks to the indexed address ($NAME,#1) in partition 1. Operand 2 must be a word value.

```
MOVE    ($NAME,#1),C'  ',(4,WORD),TKEY=0
```

14) Move the leftmost byte value X'00' to B when assembling with $EDXASM. Move the rightmost byte value X'02' to B when assembling with the macro assemblers. A has a value of X "0002"

```
A       EQU   2
         .
         .
         .
        MOVE    B,+A,(1,BYTE)
```

15) Move the four-byte character string '3333' to the indexed address (HERE,#1) in partition 1.

```
MOVE    (HERE,#1),C'3',(4,BYTE),TKEY=0
```

16) Move the character string '22222222' to the indexed address (HERE,#1) in partition 1.

```
MOVE    (HERE,#1),C'12',(8,BYTE),TKEY=0
```

Only one character may be specified in immediate mode. When assembled with the macro assembler the system takes the rightmost character. In this example the character string has been truncated and 8 characters of 2 have been moved.

17) Move the data string X'0505050505' to the indexed address (THERE,#1) in partition 1.

```
MOVE    (THERE,#1),X'05',(5,BYTE),TKEY=0
```

MOVEA - Move an address

The MOVEA instruction moves the address of operand 2 to operand 1.

Syntax:

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

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

Operand	***Description***
opnd1	The label of the data area to receive the address of opnd2. This operand must be a word in length.
opnd2	The label of the data area whose address is moved to opnd1.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) Move the address of A into PTR.

```
MOVEA   PTR,A
```

2) Move the address of B plus 4 bytes into PTR.

```
MOVEA   PTR,B+4
```

MULTIPLY - Multiply integer values

The MULTIPLY instruction multiplies an integer value in operand 1 by an integer value in operand 2. The values can be positive or negative. To multiply floating-point values, use the FMULT instruction.

See the DATA/DC statement for a description of the various ways you can represent integer data.

The supervisor places X'80000000' in the first two words of the task control block if an overflow condition occurs during double-precision multiplication.

Note: You can abbreviate the instruction as MULT.

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
```

Operand	***Description***
opnd1	The label of the data area containing the value to be multiplied by opnd2. Opnd1 cannot be a self-defining term. The system stores the result of the MULTIPLY operation in opnd1 unless you code the RESULT operand.
opnd2	The value by which opnd1 is multiplied. You can specify a self-defining term or the label of a data area. The value of opnd2 does not change during the operation.
count	The number of consecutive values in opnd1 on which the operation is to be performed. The maximum value allowed is 32767.
RESULT=	The label of a data area or vector in which the result is placed. The variable you specify for opnd1 is not changed if you specify RESULT. This operand is optional.
PREC=xyz	Specify the precision of the operation in the form xyz, where x is the precision for opnd1, y is the precision for opnd2, and z is the precision of the result ("Mixed-precision Operations" on page LR-283 shows the precision combinations allowed for the MULTIPLY instruction). You can specify single precision (S) or double precision (D) for each operand. Single precision is a word in length; double precision is two words in length. The default for opnd1, opnd2, and the result is single precision.

MULTIPLY - Multiply integer values *(continued)*

If you code a single letter for PREC, the letter applies to opnd1 and the result. Opnd2 defaults to single precision. If, for example, you code PREC=D, opnd1 and the result are double precision and opnd2 defaults to single precision.

If you code two letters for PREC, the first letter applies to opnd1 and the result, and the second letter applies to opnd2. With PREC=DD, for example, opnd1 and the result are double precision and opnd2 is double precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Mixed-precision Operations

The following table lists the precision combinations allowed for the MULTIPLY instruction:

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

Syntax Examples

1) Multiply a value in C by a value in D. The result of the operation is double precision.

```
MULT    C,D,RESULT=E,PREC=SSD
```

2) Multiply a double-precision value in A by 10. The result of the operation is double precision.

```
MULT    A,10,PREC=D
```

3) Multiply the single-precision values at X and X+2 by 10.

```
MULTIPLY    X,10,2
```

MULTIPLY - Multiply integer values *(continued)*

Coding Example

The MULTIPLY instruction at label M1 multiplies a full-word value in the data area labeled HOURS by 60. The instruction places the result in the data area labeled MINUTES. MINUTES is defined with the P2= parameter naming operand on the MULTIPLY instruction labeled M2.

At label M2, the second operand, defined with the parameter naming operand P2=, is multiplied by the value located at label SIXTY. The result is placed in the data area labeled SECONDS.

The first pair of MULTIPLY instructions uses the single-precision default for opnd1, opnd2, and RESULT=.

The third MULTIPLY instruction, at M3, multiplies the doubleword value at label MILLISEC by 1000, and places the doubleword result in MILLISEC.

The last MULTIPLY instruction, at label M4, multiplies the value at label OP11 by the value at label OP12, and places the result in the data area labeled RESULTX. Because the count operand equals 2, this instruction also multiplies the value at label OP21 by the value at label OP12, and places the result at RESULTX+2.

```
         .
         .
         .
M1       MULTIPLY HOURS,60,RESULT=MINUTES
M2       MULT     SIXTY,0,RESULT=SECONDS,P2=MINUTES
         MOVE     MILLISEC,0
         MOVE     MILLISEC+2,SECONDS
M3       MULT     MILLISEC,MILLI,PREC=DSD
         .
M4       MULTIPLY OP11,OP12,2,RESULT=RESULTX
         .
         .
HOURS    DATA     F'0'
SECONDS  DATA     F'0'
SIXTY    DATA     F'60'
MILLISEC DATA     D'0'
MILLI    DATA     F'1000'
OP11     DATA     F'1'
OP21     DATA     F'2'
OP12     DATA     F'3'
RESULTX  DATA    2F'0'
         .
         .
         .
```

NETCTL - Controlling SNA message exchange

The NETCTL instruction controls the exchange of status and error information between your Series/1 application program and the host program.

You can use the instruction to:

- Send error or status messages to the host application program
- Receive error or status messages from the host application program.

Before you can use the NETCTL instruction, you must establish a session with the host. You can use NETCTL to receive status information regardless of which session partner has the right-to-send.

Syntax:

```
 label        NETCTL    LU=,BUFF=,TYPE=,EXIT=,
                        P1=,P2=

Required:    LU=
Defaults:    TYPE=RECV
Indexable:   none
```

Operand	*Description*
LU=	Identifies the session logical unit (LU) number (from 1–32).
BUFF=	The label of a 6-byte status area that is used when you code TYPE=RECV, TYPE=REJECT, or TYPE=LUSTAT.

If you do not specify RECV, REJECT, or LUSTAT for the TYPE operand, the BUFF operand is ignored. The use of the status area is as follows:

- If you specify TYPE=RECV, the status received from the host is placed in this area. The format of the status information varies depending on what type of information it is. The NETCTL return codes indicate the type of status information received.

  If the return code indicates message reject, status message, or request for right-to-send, the status area is as follows:

  Message reject— The first two bytes of the area are the system sense code. The next two bytes are the user sense code.

NETCTL - Controlling SNA message exchange *(continued)*

If you do not select message resynchronization support for the session, the last two bytes are the message number of the message rejected by the host. If you do select message resynchronization support for the session, the message rejected by the host is always the last message sent.

Status message— The first two bytes of the area are the status value. The next two bytes are the status extension field.

Request for right-to-send— The first two bytes of the area are the signal value. The next two bytes are the signal extension field.

- If you specify TYPE=REJECT, you must supply the sense codes indicating the reason the host message is unacceptable. The first two bytes of the area are the system sense code. The next two bytes are the user sense code. If you do not specify the sense codes,the host receives a system sense code of X'081C' (Request Not Executable) along with a user sense code of X'0000' (No-operation).

  The host message rejected is always the last message received from the host.

- If you specify TYPE=LUSTAT, you must supply the status codes to be sent to the host. The first two bytes of the area are the status value. The next two bytes are the status extension field.

TYPE= The control operation to be performed. Code one of the following:

RECV Receive status information from the host. The return code indicates the type of status information received. If applicable, the area specified in the BUFF operand receives data associated with the status. RECV is the default.

ACCEPT Send the host a message acceptance, if necessary, for the message received.

REJECT Send the host a message rejection for the message received. The sense code, containing the reason for the rejection, is returned in the area specified in the BUFF operand.

CANCEL Cancel a partially transmitted message.

QEC Ask the host to temporarily stop transmitting messages after the current message.

RELQ Ask the host to resume sending messages. This operand is valid only if you have previously issued TYPE=QEC.

SIG Ask the host to give the right-to-send to the Series/1 SNA application.

NETCTL - Controlling SNA message exchange *(continued)*

LUSTAT Send status information to the host. The 4-byte status code to be sent is contained in the area you specified with the BUFF operand.

RTR Notify the host that the SNA application is ready to receive the next message.

The BUFF parameter is required if TYPE=RECV, REJECT, or LUSTAT.

EXIT= The label of the error-processing routine for your program. Control passes to this label if any return code other than -1 is returned to your program.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

The examples presented here illustrate various ways in which you can use the NETCTL instruction to control the exchange of messages.

1) Receiving Status from Host

This example shows the use of a NETCTL instruction to receive status information from the host program. The location STATUS receives the status data (if any).

```
          NETCTL    LU=NETLU,TYPE=RECV,BUFF=STATUS
            .
            .
            .
NETLU     DATA    F'1'
STATUS    DATA    F'6'
```

2) Rejecting a Message

This example shows a NETCTL instruction that rejects a message received from the host program.

```
          NETCTL    LU=NETLU,TYPE=REJECT
            .
            .
            .
NETLU     DATA    F'1'
```

NETCTL - Controlling SNA message exchange *(continued)*

3) Sending Status to Host

In this example, a NETCTL instruction sends status information to the host program. The location STATUS receives the status data.

```
          NETCTL    LU=NETLU,TYPE=LUSTAT,BUFF=STATUS
             .
             .
             .
NETLU     DATA    F'1'
STATUS    DATA    F'6'
```

Return Codes

The NETCTL return codes are placed in the first word of the task control block ($TCBCO) of the task that issued the instruction.

The positive return codes from NETCTL TYPE=RECV contain bit-significant values to allow for efficient analysis in the Series/1 SNA application. The bit positions have the following meaning:

....1	End of transaction received
....1.	Right-to-send received

The following values are returned in combination with the above bit-significant information:

X'0010'	Status message received
X'0020'	Message being received from host canceled
X'0030'	Session termination request received
X'0050'	Request for right-to-send received
X'0060'	Host permission to resume sending received
X'0070'	Message sent to host rejected

NETCTL - Controlling SNA message exchange *(continued)*

The valid combinations of the values and bit positions are listed in the following decimal return codes.

Code	Condition
112	Negative response received
096	RELQ received
080	SIGNAL received
048	SHUTDOWN received
034	CANCEL with CD received
033	CANCEL with EB received
032	CANCEL received
018	LUSTAT with CD received
017	LUSTAT with EB received
016	LUSTAT received
002	CHANGE DIRECTION received
001	END BRACKET received
-1	Operation successful
-09	LU is busy with another operation
-10	Session does not exist
-11	Instruction must be issued under program linked to $NETCMD
-12	Invalid LU number
-13	Invalid request
-14	SNA system error
-15	NETTERM in progress
-16	Session abnormally ended by host
-17	Status available
-18	Session quiesced
-19	$SNA never loaded
-20	UNBIND HOLD received
-21	More than two tasks already running under this LU
-22	Session reset; CLEAR and STD commands received.
-25	Not right-to-send
-26	No status available

NETGET - Receive messages from the SNA host

The NETGET instruction allows your application to receive messages from the host application program. Before you can use the NETGET instruction, you must establish a logical-unit-to-logical-unit session.

When you issue the NETGET instruction, Series/1 SNA passes messages received from the host's application program into a buffer area provided by NETGET. If the buffer area is not large enough to contain the complete message, you can issue additional NETGET instructions.

NETGET supplies a return code when it receives the complete message.

Syntax:

```
label          NETGET   LU=,BUFF=,BYTES=,RECLEN=,
                        EXIT=,P1=,P2=,P3=,P4=

Required:      LU,BUFF,BYTES,RECLEN
Defaults:      none
Indexable:     none
```

Operand	*Description*
LU=	Identifies the session logical unit (LU) number (from 1–32).
BUFF=	The buffer area where the message or partial message is to be received.
BYTES=	A word value containing the length, in bytes, of the buffer area you specified in the BUFF operand.
RECLEN=	A word value to receive the actual length, in bytes, of the message or partial message received.
EXIT=	The label of the error-processing routine for your program. Control passes to this label if a return code other than -1 is returned to your application.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

NETGET - Receive messages from the SNA host *(continued)*

Syntax Example

This example issues a NETGET instruction to receive a message or partial message stored at address INBUFF. In addition:

- The LU is number 1 at location NETLU.
- The length of the input area is at location INBLEN.
- The length of the message or partial message received is stored at location COUNT.

```
          NETGET   LU=NETLU,BUFF=INBUFF,BYTES=INBLEN,                    C
                RECLEN=COUNT
            .
            .
            .
NETLU     DATA   F'1'
INBUFF    DATA   XL80
INBLEN    DATA   F'80'
COUNT     DATA   F'0'
```

Return Codes

The NETGET return codes are placed in the first word of the task control block ($TCBCO) of the task that issued the instruction.

The positive return codes from NETGET contain bit-significant values to allow for efficient analysis in the Series/1 SNA application. The bit positions have the following meaning:

....1	Function management header received
....1.	End of message received
.... .1..	Right-to-send received
.... 1...	Response to message requested
....1	End of transaction received
....1.	Start of transaction received

NETGET - Receive messages from the SNA host *(continued)*

The valid combinations of the bit positions are listed in the following decimal return codes:

Code	Condition
059	Start and end of transaction, end of message and FMH received, response requested
058	Start and end of transaction, and end of message received response requested
051	Start and end of transaction, end of message and FMH received
050	Start and end of transaction, and end of message received
047	Start of transaction, end of message, FMH, and right-to-send received, response requested
046	Start of transaction, end of message, and right-to-send received, response requested
043	Start of transaction, end of message, and FMH received, response requested
042	Start of transaction, end of message, and response requested
039	Start of transaction, end of message, FMH, and right-to-send received
038	Start of transaction, end of message, and right-to-send received
035	Start of transaction, end of message, and FMH received
034	Start of transaction and end of message received
033	Start of transaction and FMH received
032	Start of transaction received

NETGET - Receive messages from the SNA host *(continued)*

NETGET Return Codes (Continued)

Return Code	Condition
027	End of transaction, end of message and FMH received, response requested
026	End of transaction and end of message received, response requested
019	End of transaction, end of message and FMH received
018	End of transaction and end of message received
015	End of message, FMH, and right-to-send received, response requested
014	End of message, and right-to-send received, response requested
011	End of message, and FMH received, response requested
010	End of message received, response requested
007	End of message, FMH, and right-to-send received
006	End of message and right-to-send received
003	End of message and FMH received
002	End of message received
001	FMH received
-1	Operation successful
-09	LU is busy with another operation
-10	Session does not exist
-11	Instruction must be issued under program linked to $NETCMD
-12	Invalid LU number
-13	Invalid request
-14	SNA system error
-15	NETTERM in progress
-16	Session abnormally ended by host
-17	Status available
-19	$SNA never loaded
-20	UNBIND HOLD received
-21	More than two tasks already running under this LU
-22	Session reset; CLEAR and STD commands received.
-25	No messages available
-26	Host initiated transaction

NETHOST - Build an SNA host ID data list

The NETHOST instruction generates an assembly-time host ID data list that defines logical unit (LU) requirements and session resources.

Certain operands in the NETHOST instruction can affect the performance of other LU operations. You may, therefore, need the help of the host system programmer when coding the instruction. You also may require the host system programmer's knowledge of SNA protocols.

Syntax:

```
label        NETHOST  ISAPPID=,ISMODE=,ISPASWD=,ISQUEUE=,
                      ISRQID=,ISUSFLD=,SSCPID=

Required:    ISAPPID=,ISMODE=
Defaults:    ISPASWD=,ISRQID=,ISUSFLD= (all default to 8 blanks)
             ISQUEUE=NO,SSCPID=6X'00' (bytes)
Indexable:   none
```

Operand	*Description*
ISAPPID=	A 1–8 character name that identifies the host user program identification (APPLID) to be used for a session. Trailing blanks are ignored by NETINIT.
ISMODE=	A 1–8 character name that identifies the set of rules and protocol for a session. The system services control point (SSCP) also uses the name to build the CINIT request.
ISPASWD=	A password of 1–8 characters used to verify the identity of a Series/1 user. The default of eight blanks causes NETINIT to generate a null (zero length) field in the INITSELF command. NETINIT ignores trailing blanks.
ISQUEUE=	YES, to place the ITITSELF request in a queue if it cannot be executed immediately. NO (the default), to prevent the request from being held in a queue.
ISRQID=	The 1–8 character name that identifies the Series/1 user initiating a request. You can also use ISRQID to establish authority for you to use a particular resource. The default of eight blanks causes NETINIT to generate a null (zero length) field in the INITSELF command. NETINIT ignores trailing blanks.

NETHOST - Build an SNA host ID data list *(continued)*

ISUSFLD= A 1–20 character string for carrying data you specify. Network services request processors do not process this data. The Series/1 SNA support passes the data to the primary logical unit (PLU). The default of eight blanks causes NETINIT to generate a null (zero length) field in the INITSELF command. NETINIT ignores trailing blanks.

SSCPID= The system services control point (SSCP) identification for the network to be attached. You can code this operand using 0–12 hexadecimal digits. A 0 value specifies the session is to be opened with any SSCP attached.

Specify any 6-byte binary value. However, to be meaningful, the bit representation must match the identification of the attached SSCP. The default is 6 bytes of zeros.

NETINIT - Establish an SNA session

The NETINIT instruction initiates a request for establishing a session with the host application program. The established session remains in effect until you end it by issuing a NETTERM instruction.

Note: In coding your program, you can (if the system resources are available) establish multiple sessions for each task. All tasks using these sessions must be within the same program.

Syntax:

```
label        NETINIT   LU=|HOLDLU=,HOSTID=,MSGDATA=,
                       SESSPRM=,ATTNEV=,RDSCB=,ERRCODE=,
                       FULLDPX=,ACQUIRE=,RESYNC=,RTYPE=,
                       EXIT=,P1=,P2=,P3=,P4=,P5=,P6=

Required:    LU=|HOLDLU=,HOSTID=
Defaults:    ACQUIRE=YES,RESYNC=YES,RTYPE=DISK,FULLDPX=NO
Indexable:   none
```

Operand	*Description*
LU=	Identifies the session logical unit (LU) number. You must code the label of a value from 0–32. If you code a value of zero, the Series/1 SNA support assigns the next available logical LU number and places the number in the second word of the task control block ($TCBC02) for your SNA application. If you specify this operand, you cannot specify the HOLDLU operand on this instruction.
HOLDLU=	The session LU number to be reestablished after receiving an UNBIND HOLD. You must code the label of a value from 0–32. If you specify this operand, you cannot specify the LU operand on this instruction.
HOSTID=	The label of the NETHOST data definition.
MSGDATA=	The label of a 6-byte data area where the SNA support stores information about messages exchanged during the session. If RESYNC=YES or INIT, the following considerations apply: • If RTYPE=DISK, MSGDATA is ignored.

NETINIT - Establish an SNA session *(continued)*

- If RTYPE=STG, MSGDATA is required. SNA uses the data area you specify with MSGDATA for resynchronization data. SNA returns the resynchronization data on successful completion of an SNA operation. The resynchronization data is reserved for SNA use only and must be supplied on the NETINIT instruction when the session is restarted.

If RESYNC=NO, MSGDATA is optional. When you specify MSGDATA, SNA uses the area to hold message data. When a NETPUT LAST=YES operation is successful, SNA stores the number assigned to the message sent to the host in the first and second bytes of the data area. The remaining bytes of the area are reserved for SNA use only.

SESSPRM= The label of a data area where SNA stores session-establishment parameters (BIND) received from the host. The area contains the parameters after the NETINIT operation completes successfully. This area must be 256 bytes.

ATTNEV= The address of an event control block (ECB) to be posted when an attention event occurs while no SNA operations are active. You should issue a NETGET instruction to determine whether the event is for status information or data.

RDSCB= The address of an opened data set control block (DSCB) to be used by SNA resynchronization processing. Code this operand only if you specify RTYPE=DISK.

ERRCODE= The label of a 4-byte data area where SNA stores extended error information. If you code this operand and the SNA operation returns a negative return code (other than -1), this data field identifies the SNA instruction and the related SNA function that failed, plus the return code of the SNA function. A breakdown of the data area follows:

- Byte 1— The SNA operation in progress when the error was encountered:

  00 - NETINIT
  01 - NETPUT
  02 - NETGET
  03 - NETCTL
  05 - NETTERM

- Byte 2— The Event Driven Executive or SNA base function that reported the error. The following hexadecimal codes are returned:

  01 - NETOPEN
  02 - NETRECV
  03 - NETSEND
  04 - NETCLOSE
  05 - NETBIND
  06 - NETUBND
  08 - BIND event post code

NETINIT - Establish an SNA session *(continued)*

0A - READ
0B - WRITE
0C - Session termination

Note: Refer to *IBM Series/1 Event Driven Executive Systems Network Architecture and Remote Job Entry Guide*, SC34-0402, for additional information on the return codes for these functions.

- Bytes 3 and 4— The error return code from the Event Driven Executive or SNA base function.

FULLDPX= NO (the default), to establish a session in a transmission mode of half duplex.

YES, to establish a session in a transmission mode of duplex.

Note: If you code FULLDPX=YES, you cannot use message resynchronization and attention event processing.

ACQUIRE= YES (the default), to cause SNA to initiate the session for your application program.

NO, to indicate that the host is to initiate the session.

RESYNC= NO, to disable session resynchronization.

YES (the default), to use the contents of the resynchronization data set during session establishment.

INIT, to initialize the contents of the resynchronization data set during session establishment.

RTYPE= DISK (the default), to save session resynchronization data on disk. You must code the RDSCB operand if you specify this parameter.

STG, to save session resynchronization data in storage. You must code the MSGDATA operand if you specify this parameter.

This operand is ignored if you code RESYNC=NO.

Note: Your program must open and close the 256-byte resynchronization data set.

EXIT= The label of the error-processing routine for the Series/1 application. Control passes to this label if a return code other than -1 is returned to your program.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

NETINIT - Establish an SNA session *(continued)*

Syntax Examples

The examples presented here illustrate various ways in which you can use the NETINIT instruction to establish a session.

1) Session with Resynchronization Data to Disk

This example illustrates establishing a session where the resynchronization data resides on a disk. In addition:

- The LU is number 1 at location NETLU.
- Series/1 SNA initiates the session with the host. SNA saves the extended error information at location SAVERC.
- The resynchronization data set RDSCB is RESTART.

```
         NETINIT  LU=NETLU,HOSTID=SNAID,ACQUIRE=YES,                  C
               ERRCODE=SAVERC,RESYNC=YES,RTYPE=DISK,                  C
               RDSCB=RESTART
            .
            .
            .
NETLU    DATA     F'1'
SAVERC   DATA     4F'0'
RESTART  DSCB     DS#=RSYNC,DSNAME=RSYNDSCB
SNAID    NETHOST  ISAPPID=IMS,ISMODE=INQUIRY
```

2) Session with Resynchronization Data to Storage

This example illustrates establishing a session where the resynchronization data resides in storage. In addition:

- Series/1 SNA support waits for the host to initiate the session.
- SNA initializes the contents of the resynchronization data set when the session starts.
- SNA saves the resynchronization data at address RDATA.

```
         NETINIT  LU=NETLU,HOSTID=SNAID,ACQUIRE=NO,                   C
              RESYNC=INIT,RTYPE=STG,MSGDATA=RDATA
            .
            .
            .
NETLU    DATA     F'1'
RDATA    DATA     6F'0'
SNAID    NETHOST  ISAPPID=CICS,ISMODE=INQUIRY
```

NETINIT - Establish an SNA session *(continued)*

3) Session without Resynchronization

This example illustrates establishing a session without resynchronization support. SNA saves the message numbers at address MDATA.

```
          NETINIT  LU=NETLU,HOSTID=SNAID,ACQUIRE=NO,                   C
                RESYNC=NO,MSGDATA=MDATA
            .
            .
            .
NETLU     DATA     F'1'
MDATA     DATA     6F'0'
SNAID     NETHOST  ISAPPID=JES2,ISMODE=RMT26
```

NETINIT - Establish an SNA session *(continued)*

Return Codes

NETINIT return codes are placed in the first word of the task control block ($TCBCO) of the task that issued the instruction.

If you code the ERRCODE operand on the NETINIT instruction, additional error information is returned, when appropriate, to the area you specified. Refer to *IBM Series/1 Event Driven Executive Systems Network Architecture and Remote Job Entry Guide*, SC34-0402 for a description of this extended error code information.

The positive return codes from NETINIT contain bit-significant values to allow for efficient analysis in the Series/1 SNA application. For a description of the bit-significant values, refer to *IBM Series/1 Event Driven Executive Systems Network Architecture and Remote Job Entry Guide*, SC34-0402.

The decimal return codes that could be returned from a NETINIT operation follow.

Code	Condition
081	Message flow to host cold-started, message to host possibly lost Message flow from host cold-started, no messages from host lost
049	Message flow to host cold-started, message to host lost Message flow from host cold-started, no messages from host lost
032	Message to host lost
019	Message flow to host cold-started Message flow from host cold-started, message from host lost
017	Message flow to host cold-started, no messages to host lost. Message flow from host cold-started, no messages from host lost
004	Partially presented message from host lost
002	Unpresented message from host lost
-1	Operation successful
-12	Invalid LU number
-14	SNA system error
-15	NETTERM in progress
-16	Session abnormally ended by host
-19	$SNA never loaded
-26	Logical unit already open
-27	No logical unit available
-30	BIND from host rejected
-31	STSN error
-32	No NETTERM HOLD=YES issued

NETPUT - Send messages to the SNA host

The NETPUT instruction transmits messages from a Series/1 application program to the host application program. You can issue a NETPUT instruction only after establishing a session successfully.

You can send a complete message to the host with one NETPUT operation, or, if necessary, you can send a single message with multiple NETPUT operations.

You must have the right-to-send for the NETPUT operation to be successful. If you are receiving and need to send, issue the NETCTL instruction with TYPE=SIG to request the right-to-send. When no transaction is active on the session, both you and the host have the right-to-send.

You can cancel a message during transmission to the host by issuing a NETCTL instruction with TYPE=CANCEL. The host discards any part of the message it has already received. See the NETCTL instruction for more coding information.

Syntax:

```
label        NETPUT   LU=,BUFF=,BYTES=,EOT=,FMH=,INVITE=,
                      LAST=,VERIFY=,EXIT=,P1=,P2=,P3=

Required:    LU=,BUFF=,BYTES=
Defaults:    EOT=NO,FMH=NO,INVITE=YES,
             LAST=YES,VERIFY=NO
Indexable:   none
```

Operand	***Description***
LU=	Identifies the session logical unit (LU) number. You must code the label of a value from 1–32.
BUFF=	The message, or partial message, to be sent.
BYTES=	A word containing the number of bytes in the message or partial message to be sent.
EOT=	YES, to end the transaction after the message is sent. NO (the default), to avoid ending the transaction after the message is sent. This operand is only recognized on the first NETPUT instruction you issue for a message.
FMH=	YES, if the message contains function management (FM) headers.

NETPUT - Send messages to the SNA host *(continued)*

NO (the default), if the message does not contain FM headers.

This operand is only recognized on the first NETPUT instruction you issue for a message.

INVITE= YES (the default), to give the host the right-to-send after this message is transmitted.

NO, if you do not want to give the host the right-to-send.

This operand is ignored unless you specify LAST=YES (see the LAST operand).

LAST= YES (the default), if this is the last NETPUT operation for the message.

NO, if this is not the last NETPUT operations for the message.

VERIFY= YES, if the host should verify that it received the message.

NO (the default), if you do require verification.

This operand is ignored if you do not specify LAST=YES.

EXIT= The label of the error-processing routine for the Series/1 application. Control passes to this label if any return code other than -1 is returned to your application.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

The examples presented here illustrate various ways you can use the NETPUT instruction to send messages.

1) Sending a Message with a Single NETPUT

This example illustrates sending a message to the host using one NETPUT instruction. In addition:

- The LU is number 1 at location NETLU.
- The message to be sent is at location OUTBUFF.
- The length of the message to be sent is at location BYTECNT.
- The data is to be sent as a complete message.

NETPUT - Send messages to the SNA host *(continued)*

- The host receives the right-to-send.
- Function management headers are included in the data.

```
          NETPUT    LU=NETLU,BUFF=OUTBUFF,BYTES=BYTECNT,             C
                INVITE=YES,FMH=YES,LAST=YES
              .
              .
              .
NETLU     DATA  F'1'
OUTBUFF   DATA  CL80'MESSAGE'
BYTECNT   DATA  F'80'
```

2) Sending a Message with Multiple NETPUT Operations

This example illustrates one message being sent to the host with three NETPUT instructions. In addition:

- The lengths of the "partial messages" to be sent are at locations BYTECNT1, BYTECNT2, and BYTECNT3.
- The host should verify that it received the message.
- The transaction ends after sending the message.

```
          NETPUT    LU=NETLU,BUFF=OUTBUFF1,BYTES=BYTECNT1,           C
                 EOT=YES,LAST=NO

          NETPUT    LU=NETLU,BUFF=OUTBUFF2,BYTES=BYTECNT2,           C
                 LAST=NO

          NETPUT    LU=NETLU,BUFF=OUTBUFF3,BYTES=BYTECNT3,           C
                 VERIFY=YES,LAST=YES
              .
              .
              .
NETLU     DATA  F'1'
OUTBUFF1  DATA  CL40'MESSAGE PART 1'
OUTBUFF2  DATA  CL20'MESSAGE PART 2'
OUTBUFF3  DATA  CL20'MESSAGE PART 3'
BYTECNT1  DATA  F'40'
BYTECNT2  DATA  F'20'
BYTECNT3  DATA  F'20'
```

NETPUT - Send messages to the SNA host *(continued)*

Return Codes

NETPUT return codes are placed in the first word of the task control block ($TCBCO) of the task that issued the instruction.

The positive return codes from NETPUT contain bit-significant values to allow for efficient analysis in the Series/1 SNA application. The bit positions have the following meaning:

....1	Host attempted to start a transaction

The valid combinations of the bit positions are listed in the following decimal return codes:

Return Code	Condition
001	Host attempted to start transaction
-1	Operation successful
-09	LU is busy with another operation
-10	Session does not exist
-11	Instruction must be issued under program linked to $NETCMD
-12	Invalid LU number
-13	Invalid request
-14	SNA system error
-15	NETTERM in progress
-16	Session abnormally ended by host
-17	Status available
-18	Session quiesced
-19	$SNA never loaded
-20	UNBIND HOLD received
-21	More than two tasks already running under this LU. Limit is two tasks.
-22	Session reset; CLEAR and STD commands received.
-25	Not right-to-send

NETTERM - End an SNA session

The NETTERM instruction releases the logical communications path previously established between session partners with the NETINIT instruction. NETTERM ends the session and releases the Series/1 resources used for the session.

You can use the system resources freed with the NETTERM instruction to establish other sessions.

At any time, either the host program or your application program can end the session.

Syntax:

```
label         NETTERM  LU=,HOLD=,EXIT=,P1=

Required:     LU=
Defaults:     HOLD=NO
Indexable:    none
```

Operand	*Description*
LU=	Identifies the session logical unit (LU) number. You must code a label pointing to a value from 1–32.
HOLD=	YES, to keep session resources if the host issues a BIND command following the NETTERM instruction. NO (the default), to end the session and release all session resources. Code this operand only when the host issues an UNBIND HOST command.
EXIT=	The label of the error-processing routine for your program. Control passes to this label if any return code other than -1 is returned to your application.
Px=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

The following example shows the use of the NETTERM instruction to end a session. The LU address for the ended session is at address NETLU.

NETTERM - End an SNA session *(continued)*

```
              NETTERM LU=NETLU
                 .
                 .
                 .
NETLU         DATA    F'1'
```

Return Codes

The NETTERM return codes are placed in the first word of the task control block ($TCBCO) of the task that issued the instruction.

The positive return codes from NETTERM contain bit-significant values to allow for efficient analysis in the Series/1 SNA application. The bit positions have the following meaning:

....1	Message from host rejected during termination
....1.	Message to host rejected during termination
.... .1..	Message to host aborted during termination
.... 1...	Message from host aborted during termination

The valid combinations of the bit positions are listed in the following decimal return codes:

Return Code	Condition
009	CANCEL received during NETTERM and negative response sent during NETTERM
008	CANCEL received during NETTERM
007	CANCEL sent during NETTERM and negative response received during NETTERM and negative response sent during NETTERM
006	CANCEL sent during NETTERM and negative response received during NETTERM
005	CANCEL sent during NETTERM and negative response sent
004	CANCEL sent during NETTERM
003	Negative response received during NETTERM and negative response sent during NETTERM
002	Negative response received during NETTERM
001	Negative response sent during NETTERM
-1	Operation successful
-10	Session does not exist
-11	Instruction must be issued under program linked to $NETCMD
-12	Invalid LU number
-14	SNA system error
-15	NETTERM in progress
-16	Session abnormally ended by host
-19	$SNA never loaded
-20	UNBIND HOLD received
-25	No UNBIND HOLD received

NEXTQ - Add entries to a queue

The NEXTQ instruction allows you to add entries to a queue defined with the DEFINEQ statement. The system removes a queue entry from the free chain of the queue and places the entry in the queue's active chain.

Syntax:

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

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

Operand	*Description*
qname	The name of the queue in which to place the entry. The queue name is the label of the DEFINEQ statement that creates the queue.
loc	The label of a 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=	The label of the first instruction of the routine to be invoked if a "queue full" condition is detected during the execution of this instruction. If you do not specify this operand, control returns to the next instruction after the NEXTQ. A return code of -1 in the first word of the task control block indicates that the operation completed successfully. A return code of +1 indicates that the queue is full.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

NEXTQ - Add entries to a queue *(continued)*

Coding Examples

1) The following example uses each of the queuing instructions. The program defines a queue area that contains four six-word buffers. The FIRSTQ instruction obtains the oldest entry in TIMEBUF. The GETTIME instruction obtains the date and time and updates the contents of the entry obtained by FIRSTQ. The program stores the new time and date in TIMEQ1 and TIMEQ2. When all buffers are allocated, the queue entries are printed on a first-in-first-out basis, then on a last-in-first-out basis, and the buffers used are freed. Each queue instruction executes 8 times.

```
QTEST     PROGRAM   START
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
```

NEXTQ - Add entries to a queue *(continued)*

2) In this example, index register 1 points to a block of storage in a buffer area. The NEXTQ instruction places the address of that location (contained in register #1) into the queue defined by the QUE1 label. If the queue is full, the program branches to the FULLQUE1 label. Otherwise, the MOVE instruction places 32 bytes of data, beginning at the address labeled DATAREC, into the buffer area. The ADD instruction updates #1 so that it points to the next sequential block of storage in the buffer.

```
         SUBROUT NEXTQUE1
*
         NEXTQ   QUE1,#1,FULL=FULLQUE1
         MOVE    (0,#1),DATAREC,(32,BYTES)
         ADD     #1,32
         RETURN
*
FULLQUE1 EQU      *
         PRINTEXT '@QUE1 QUEUE BUFFER FULL'
         GOTO     ENDIT
           .
           .
QUE1     DEFINEQ  COUNT=8
           .
           .
ENDIT    EQU      *
         PROGSTOP
DATAREC  DATA     16F'0'
```

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
1	Queue is full

NOTE - Store next-record pointer

The NOTE instruction 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 instruction.

Syntax:

```
label          NOTE      DSx,loc,PREC=,P2=

Required:      DSx,loc
Defaults:      PREC=S
Indexable:     loc
```

Operand	*Description*
DSx	Code DSx, where "x" is the relative position (number) of a data set in the list of data sets you define on 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 used in place of DSx.
loc	This operand specifies the address of a fullword or doubleword of storage that will contain the next-record-pointer as the result of executing a NOTE instruction. This value can be used as the relative record number (relrecno) in a subsequent POINT or direct READ or WRITE operation. When this operand is coded as an indexable value or as an address label, the PREC operand can be used to further define whether relrecno is to be a single-word or double-word value. If the PREC operand is coded as PREC=D, then the range of relrecno is extended beyond the 32767 value to the limit of a double-word value.
PREC=	This optional operand further defines the relrecno operand only when relrecno is coded as an address or as an indexable value. The default value is S and has the same effect on relrecno as coding PREC=S. That effect is to limit the value of relrecno to single-word precision or a value of X'7FFF' (32767). Coding PREC=D gives a double-word precision attribute to the relrecno operand and, therefore, extends its maximum value range to a double-word value.
P2=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

NOTE - Store next-record pointer *(continued)*

Syntax Examples

1) The following NOTE instruction is valid for records that do not exceed a length of 32,767.

```
NOTEL1    NOTE     DS2,LOCS
           .
           .
LOCS      DATA     F'0'
```

2) The NOTE instruction in this example is valid for records that exceed 32,767 because the variable LOCD is double-word precision.

```
NOTEL2    NOTE     DS3,LOCD,PREC=D
           .
           .
LOCD      DATA     D'0'
```

PLOTGIN - Enter scaled cursor coordinates

The PLOTGIN instruction allows you to specify scaled cursor coordinates interactively. The instruction uses the coordinates you specify to plot curves. PLOTGIN rings the bell and displays the cross-hair cursor. It waits for you to position the cross-hairs and enter a single character. The cursor coordinates you enter are scaled with the use of the plot control clock (PLOTCB). A description of the control block follows this instruction.

Syntax:

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

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

Operand	*Description*
x	The location where the x cursor coordinate value is to be stored.
y	The location where the y cursor coordinate value is to be stored.
char	The location where the character you select is to be stored. The character is stored in the rightmost byte. The left byte is set to zero. If you do not code this operand, the instruction does not store the selected character.
pcb	Label of an 8-word plot-control block.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Plot Control Block (PLOTCB)

The plot control block defines the size and position of the plot area on the screen and the data values associated with the edges of the plot area. The PLOTCB consists of eight words of data defined by DATA statements.

You must build a PLOTCB in your graphics program when using the PLOTGIN, XYPLOT or YTPLOT instructions. The format of the control block 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'
```

PLOTGIN - Enter scaled cursor coordinates *(continued)*

You must specify an explicit value for all eight statements. The required values are defined below:

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 x 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

Syntax Example

Read x and y cursor coordinates and store them in X and Y, respectively. Store characters in the data area labeled CHAR. The plot control block is at label PCB.

```
          PLOTGIN     X,Y,CHAR,PCB
            .
            .
            .
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'
```

POINT - Set next-record pointer

The POINT instruction causes the value of a data set's next-record-pointer, which is maintained by the system, to be set to a new value. The system uses this new value in later sequential READ or WRITE operations.

Syntax:

```
label         POINT     DSx,relrecno,PREC=,P2=

Required:     DSx,relrecno
Defaults:     PREC=S
Indexable:    relrecno
```

Operand	***Description***
DSx	Code DSx, where "x" is the relative position (number) of the data set in the list of data sets you define on 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	This operand sets a new value in the system-maintained next-record-pointer. This parameter can be either a constant or the label of the value to be used. If this value is coded as a self-defining term, or an equated value which is preceded by a plus sign (+), then it is assumed to be a single-word value and is, therefore, generated as an inline operand. Because this is a one-word value, it is limited to a range of 1 to 32767. When this operand is coded as an indexable value or as an address, the PREC operand can be used to further define whether relrecno is to be a single-word or double-word value. If the PREC operand is coded as PREC=D, then the range of relrecno is extended beyond the 32767 value to the limit of a double-word value (2147483647).
PREC=	This operand further defines the relrecno operand when you code an address or an indexable value for relrecno. PREC=S (the default) limits the value of the relrecno operand to a single-precision value of 32767 (X'7FFF'). PREC=D extends the maximum range for the relrecno operand to a doubleword value of 2147483647 (X'7FFFFFFF').
P2=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

POINT - Set next-record pointer *(continued)*

Syntax Examples

1. The following POINT instruction is valid for records that do not exceed a length of 32767.

```
POINTL1   POINT   DS2,LOCS
            .
            .
LOCS      DATA    F'0'
```

2. This POINT instruction is valid for records that exceed 32767 because the variable LOCD is double-word precision.

```
POINTL2   POINT   DS3,LOCD,PREC=D
            .
            .
LOCD      DATA    D'0'
```

POST - Signal the occurrence of an event

The POST instruction signals the occurrence of an event.

A POST instruction 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 Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example of posting an event in another partition. You can find more information on cross-partition services in the *Event Driven Executive Language Programming Guide*.

Syntax:

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

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

Operand	***Description***
event	The label of an event control block (ECB) that defines the event. You must code an ECB statement in your program if you compile the program under $EDXASM. $S1ASM and the S/370 host assembler generate the ECB for the event named in the POST instruction. You do not need to code an ECB statement when using either of these macro assemblers. Process interrupts are special events that can be simulated with a POST. This is useful when one task is waiting for a process interrupt and a second task wishes to start the first, as in a program termination sequence. In this case, issue a POST PIx, where "x" is a process interrupt number from 1–99 as specified in an IODEF statement.
code	A value, other than zero, to be inserted into the control block for the event. You may want to use this value as a flag that indicates a certain condition or status. To check the code value, refer to the label of the ECB statement.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

POST - Signal the occurrence of an event *(continued)*

Coding Examples

1) The POST instruction in the following example posts the event control block labeled ECB1 when TASK1 is finished processing. TASK1 reads a record from the data set MYFILE and places the record in the buffer labeled BUF. The primary task, PRINTOUT, waits for ECB1 to be posted before it continues processing. When the POST instruction posts ECB1, the primary task enqueues the system printer and prints the first 50 bytes of the record.

```
PRINTOUT      PROGRAM     START,DS=((MYFILE,EDX40))
START         EQU         *
              ATTACH      TASK1
              WAIT        ECB1
              ENQT        $SYSPRTR
              MOVE        REC,BUF,25
              PRINTEXT    REC,SKIP=1
                .
                .
                .
              PROGSTOP
BUF           BUFFER      256,WORD
ECB1          ECB
REC           TEXT        LENGTH=50
***************************************************
TASK1         TASK        NEXT
NEXT          READ        DS1,BUF,1
              POST        ECB1
              ENDTASK
              ENDPROG
              END
```

2) The following example posts an ECB labeled ECB1 which is declared as external to the assembly module.

```
              EXTRRN       ECB1
                      .
                      .
                      .
              MOVEA        B,ECB1
              POST         *,P1=B
                      .
                      .
                      .
              END
```

PRINDATE - Display the date on a terminal

The PRINDATE instruction prints the date on a terminal. The system prints the date in the form MM/DD/YY or DD/MM/YY, depending on the option coded on the SYSTEM statement when the supervisor was generated.

Note: You must include timer support in the system and have timer hardware installed to use the PRINDATE instruction. Otherwise, a program check will occur.

The supervisor places a return code in the first word of the task control block (taskname) whenever a PRINDATE instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

Syntax:

```
label         PRINDATE

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

Operand	***Description***
none	none

3101 Display Considerations

If you are using a 3101 in block mode, it will display the output from a PRINDATE instruction according to the SET,ATTR and SET,STREAM operands of a TERMCTRL statement currently in effect. For details on these operands see "TERMCTRL - Request special terminal functions" on page LR-446.

PRINDATE - Display the date on a terminal *(continued)*

Coding Example

The following example prints the date and a message on the system printer.

```
         .
         .
         .
        ENQT       $SYSPRTR
        PRINTEXT   '@ THE DATE IS '
        PRINDATE
        DEQT
         .
         .
         .
```

The data appears in one of two formats, depending on the option coded on the DATEFMT keyword of the SYSTEM statement during system generation.

If the SYSTEM statement has DATEFMT=MMDDYY (the default), the PRINDATE instruction in the above example would produce the following result on February 25, 1984:

```
THE DATE IS 02/25/84.
```

If the SYSTEM statement has DATEFMT=DDMMYY, the result of the PRINDATE operation would be:

```
THE DATE IS 25/02/84.
```

PRINT - Control printing of a compiler listing

The PRINT statement controls the printing of the compiler listing. Because no instructions or constants are generated in the object program by this statement, it can be placed between executable instructions in your source statement data set.

A program can contain any number of PRINT statements. Each PRINT statement controls the printing of the compiler listing until another PRINT statement is encountered.

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
```

Operand	***Description***
ON	A listing is printed.
OFF	No listing is printed, except for the PRINT OFF statement itself.
GEN	The listing includes all object code generated by the compiler.
NOGEN	No object code appears with the instructions in the listing. Error messages appear regardless of NOGEN. The PRINT instruction also appears 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.

PRINT - Control printing of a compiler listing *(continued)*

Coding example

The following sample program is compiled under $EDXASM using the formatting aids PRINT, TITLE, SPACE, and EJECT. The TITLE statement places the program title, "Compiler Listing Control Demonstration," at the top of each page of the listing. PRINT OFF stops the printing of the listing, which is resumed when the system encounters the PRINT ON statement. In this case, the MOVE instruction between two PRINT statements is omitted.

The SPACE statement inserts a specified number of blank lines between instructions, improving the readability of the listing. When the EJECT statement is reached, the printer ejects the page and begins printing the next line of the listing at the top of a new page. PRINT DATA causes the hexadecimal value of the first TEXT statement to be printed out in full in the left-hand column of the listing. When the default, PRINT NODATA, is coded before the second TEXT statement, the system prints only the leftmost 8 bytes of constants.

Sample Program:

```
         TITLE     'COMPILER LISTING CONTROL DEMONSTRATION'
DEMO     PROGRAM   START
START    EQU       *
         PRINT     OFF
         MOVE      COUNT,0
         PRINT     ON
LOOP     EQU       *
         ADD       COUNT,1
         SPACE     5
         IF        (COUNT,LE,10)
            PRINTEXT MESSAGE1
            PRINTNUM COUNT
         SPACE     2
         ELSE
            IF     (COUNT,LE,20)
               PRINTEXT MESSAGE2
               PRINTNUM COUNT
            ENDIF
         ENDIF
         SPACE     4
         IF        (COUNT,GT,20)
            PRINTEXT '@TERTIARY TEST MESSAGE NUMBER '
            PRINTNUM COUNT
            PROGSTOP
         ELSE
         GOTO LOOP
         ENDIF
         EJECT
COUNT    DATA  F'0'
         PRINT DATA
MESSAGE1 TEXT     '@PRIMARY TEST MESSAGE NUMBER '
         PRINT NODATA
MESSAGE2 TEXT     '@SECONDARY TEST MESSAGE NUMBER '
         ENDPROG
         END
```

PRINT - Control printing of a compiler listing *(continued)*

Compiler Listing:

```
                                        COMPILER LISTING CONTROL DEMONSTRATION
LOC    +0   +2   +4   +6   +8               SOURCE STATEMENT


0000   0008 07D9 D6C7 D9C1 D440  DEMO      PROGRAM    START
000A   0000 00E8 0168 0000 0000
0014   016C 0000 0000 0000 0100
001E   016A 0000 0000 0000 0000
0028   0000 0000 0000 0000 0000
0032   0000
0034                             START     EQU        *
                                           PRINT      OFF
003A                             LOOP      EQU        *
003A   8032 00A4 0001                      ADD        COUNT,1



0040   90A2 00A4 000A 0056                 IF         (COUNT,LE,10)
0048   0026 00A8                              PRINTEXT  MESSAGE1
004C   0028 00A4 0001                         PRINTNUM  COUNT

0052   00A0 0068                           ELSE
0056   90A2 00A4 0014 0068                    IF        (COUNT,LE,20)
005E   0026 00C8                                 PRINTEXT  MESSAGE2
0062   0028 00A4 0001                            PRINTNUM  COUNT
0068                                          ENDIF
0068                                       ENDIF



0068   E0A2 00A4 0014 00A0                 IF         (COUNT,GT,20)
0070   8026 1E1E 7CE3 C5D9 E3C9               PRINTEXT  '@TERTIARY TEST MESSAGE NUMBER '
007A   C1D9 E840 E3C5 E2E3 40D4
0084   C5E2 E2C1 C7C5 40D5 E4D4
008E   C2C5 D940
0092   0028 00A4 0001                         PRINTNUM  COUNT
0098   0022 FFFF                              PROGSTOP
009C   00A0 00A4                           ELSE
00A0   00A0 003A                           GOTO   LOOP
00A4                                       ENDIF



                                        COMPILER LISTING CONTROL DEMONSTRATION
LOC    +0   +2   +4   +6   +8               SOURCE STATEMENT


00A4   0000                      COUNT     DATA  F'0'
                                           PRINT DATA
00A6   1E1D 7CD7 D9C9 D4C1 D9E8  MESSAGE1  TEXT       '@PRIMARY TEST MESSAGE NUMBER '
00B0   40E3 C5E2 E340 D4C5 E2E2
00BA   C1C7 C540 D5E4 D4C2 C5D9
00C4   4040
                                           PRINT NODATA
00C6   201F 7CE2 C5C3 D6D5 C4C1  MESSAGE2  TEXT       '@SECONDARY TEST MESSAGE NUMBER '
00E8   0000 0000 0000 0234 0000            ENDPROG
                                           END
```

PRINTEXT - Display a message on a terminal

The PRINTEXT instruction allows you to print or display a message on any enqueued terminal, not only the loading terminal. As the default terminal, the loading terminal requires no ENQT instruction to perform a PRINTEXT. The PRINTEXT instruction also allows you to control cursor or forms movement.

The PRINTEXT instruction generally does cursor or forms movement before writing the message to the terminal.

Output for the terminal normally accumulates in the system buffer (or user buffer, if provided). The system writes this output to the terminal when it encounters a new line character (@), a forms control operand (SKIP, LINE, or SPACES), a PROGSTOP instruction, or a DEQT instruction for a terminal.

The supervisor places a return code in the first word of the task control block (taskname) whenever a PRINTEXT instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the the end of this instruction and the READTEXT instruction and also in the *Messages and Codes*.

Syntax:

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

Required:    At least one operand from the following
             list: SKIP, LINE, SPACES, or msg
Defaults:    SKIP=0,LINE=(current line),SPACES=0,
             XLATE=YES,PROTECT=NO
Indexable:   msg,LINE,SKIP,SPACES
```

Operand	***Description***
msg	The label of a TEXT statement which defines the message to be displayed or printed, or the actual message enclosed in apostrophes. You can also code the label of a BUFFER statement. When using a BUFFER statement, you must:

- Code the buffer label on the BUFFER= operand of the IOCB statement for the terminal your program enqueues.
- Move the number of characters to be printed into the index field of the BUFFER statement (msg-4).

When you use a BUFFER statement, the system does not recognize the new line character (@), and the operation executes immediately.

The maximum line size for a terminal depends on how the TERMINAL definition statement was coded during system generation. Refer to the

PRINTEXT - Display a message on a terminal *(continued)*

TERMINAL statement in the *Installation and System Generation Guide* for information on default sizes.

SKIP= The number of lines to be skipped before the system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

PRINTEXT - Display a message on a terminal *(continued)*

XLATE= NO, to send the message to the device as is, without translation. This option might be used, for example, to send graphic control characters and data.

YES (the default), to cause translation of characters from EBCDIC to the code the terminal uses to display the message.

With a 3101 in block mode, XLATE=NO also prevents the system from inserting the attribute byte and escape sequences into the message and overrides the effects of TERMCTRL SET,STREAM=YES.

Note: For a description of 3101 escape sequences refer to *IBM 3101 Display Terminal Description*, GA18-2033.

If the terminal requires that characters be sent in mirror image and you code XLATE=NO, it is your responsibility to provide the proper bit representation. For more details on mirror image, see the *Communications Guide.*

MODE= LINE, to prevent the system from interpreting each @ character it finds in the text as a request for a new line.

For 4978, 4979, and 4980 screens accessed in STATIC mode, the coding of MODE=LINE and the SPACES operand causes protected fields to be skipped over as the data is transferred to the screen ("scatter write" operation). Protected positions do not contribute to the count. For a 3101 in block mode with a static screen, the protected fields are overwritten.

Do not code this operand if you want the system to recognize @ as a new line character.

PROTECT= YES, to write protected characters to a static screen device that supports this feature, such as an IBM 4978, 4979, 4980 and 3101 in block mode. Protected characters are displayed and cannot be typed over.

NO (the default), not to write protected characters to a static screen.

When the PRINTEXT instruction is being coded for a Series/1-to-Series/1 operation, it is recommended that this operand be coded PROTECT=YES.

CAPS= Code this operand to convert a PRINTEXT message to uppercase characters. This operand is valid only for EBCDIC data that is defined by a TEXT or BUFFER statement.

Code CAPS=Y to convert all data defined by a TEXT or BUFFER statement to uppercase characters. When specifying CAPS=Y, you must link edit your program using the autocall feature of $EDXLINK.

To convert a specific number of bytes to uppercase, code that number with the CAPS operand. Capitalization starts from the first byte of the message text. For

PRINTEXT - Display a message on a terminal *(continued)*

example, CAPS=3 capitalizes the first three bytes of data defined by the TEXT or BUFFER statement.

The count you specify should not exceed the length of the TEXT or BUFFER statement that defines the message. If the length is exceeded, the operation is still performed, but data beyond the TEXT or BUFFER statement may be modified.

When you code a value with the CAPS operand, the system does an inclusive OR (IOR) of an X'40' byte to each EBCDIC byte. (See Coding Example 3 at the end of this section). A lower-case "a" (X'81'), for example, is converted to an uppercase "A" (X'C1'). Characters already capitalized remain unchanged. The IOR operation is done before the PRINTEXT instruction executes. The data is converted to uppercase in the application program.

Notes:

1. Only CAPS=Y is valid when you use the P1= operand with this instruction.
2. Coding XLATE=NO and the CAPS operand causes an assembly error.
3. When using the 4975 printer, do not code the CAPS operand if you are using the spacing character and a space modifier to increase the spacing between printed characters. See "4975 Spacing Capabilities" on page LR-328 for details on how to use the spacing character and the space modifier. This note does not refer to the 4975-01A ASCII printer.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Buffer Considerations

When a buffer overflow condition occurs, what happens to accumulated data depends on how the system definition TERMINAL statement or IOCB statement is coded. If the TERMINAL or IOCB statement contains OVFLINE=YES, the system writes the data in the buffer to the terminal and then uses the available buffer space for overflow data.

If the TERMINAL or IOCB statement contains OVFLINE=NO, any data following a buffer overflow condition is lost. Until the system writes the buffer data to the terminal, an imbedded @ will not be recognized following a buffer overflow condition. (For details on the TERMINAL definition statement, refer to the *Installation and System Generation Guide.*)

When your program issues a PRINTEXT instruction to devices other than a 4973 or 4974, and the buffer size is equal to the line size, an extra line space can occur.

When using direct I/O or when the keyword XLATE=NO is coded, the output to a terminal is written immediately.

PRINTEXT - Display a message on a terminal *(continued)*

3101 Display Considerations

If you are using a 3101 in block mode, it will normally write an attribute byte before the output data. The attribute byte controls the characteristics of the field that it precedes. One such characteristic, intensity, can be either HIGH or LOW and the field can be either blinking or nonblinking, depending on how the SET,ATTR operand was coded on the TERMCTRL statement in effect. If no attribute byte is desired, such as when writing to an existing formatted screen, code TERMCTRL ATTR=NO before using the PRINTEXT instruction. TERMCTRL ATTR=YES should then be coded to restore the writing of attribute bytes.

When the TERMCTRL statement that is in effect is coded STREAM=NO or is allowed to default to NO by not coding this operand, terminal I/O support provides the attribute byte for you. Terminal I/O also provides escape sequences for you under this condition. For a description of 3101 escape sequences, see *IBM 3101 Display Terminal Description*, GA18-2033.

If the last TERMCTRL statement was coded SET,STREAM=YES, then the SET,ATTR operand is not considered. Under this condition, terminal I/O support does not provide any attribute bytes or escape sequences.

With either STREAM=YES or NO, translation of data from EBCDIC will be performed. See the XLATE operand description.

If you are using a 3101 in block mode, the system does not recognize a new line character (@).

Note: Do not press the SEND key on a 3101 terminal while the system is doing a PRINTEXT operation to that terminal. The SEND key can affect the data being displayed.

4975 Spacing Capabilities

The following information refers to spacing capabilities only on the 4975 printer. It does not refer to such capabilities on the 4975-01A ASCII nor any other model printer.

When using the 4975 printer in draft mode, you can increase the amount of space left between printed characters on a line by inserting special spacing characters into the TEXT or BUFFER statement that defines the PRINTEXT message.

To insert additional space between characters you must include the spacing character X‘27‘ followed by a space modifier. The space modifier defines the percentage of additional space to be included. It is a hexadecimal value in the form ‘Fx‘, where “x“ is a number from 0 to 9. The space modifier ‘F0’ adds no additional space, ‘F1’ adds 10 percent additional space, and ‘F2’ adds 20 percent additional space. ‘F9’ adds 90 percent additional space and is the maximum value that you can specify.

You must insert the spacing character and the space modifier into the TEXT or BUFFER statement at each point where you want additional space. The second coding example at the end of this section shows one way to do this operation.

PRINTEXT - Display a message on a terminal *(continued)*

All printers with the exception of the 4975-01A ASCII Printer treat X'00' as a blank. The 4975-01A ASCII Printer ignores X'00' and treats it as a null character. This may cause a spacing difference if you send X'00' in your PRINTEXT instruction.

Syntax Examples

1) Print the contents of a TEXT statement at label TEXT1.

```
PRINTEXT  TEXT1
```

2) Print the text message within quotes on a new line (the new line character @ is not printed).

```
PRINTEXT  '@START OF PROGRAM'
```

3) Add four to the current cursor position and print the contents of a text statement at label TEXT2.

```
PRINTEXT  TEXT2,SPACES=4
```

4) If not currently at the first line of a page or screen, skip to a new page and then skip two lines and print the contents of a text statement at TEXT3.

```
PRINTEXT  TEXT3,LINE=1,SKIP=2
```

5) Skip one line. If any output is residing in the system buffer or the terminal I/O buffer, the system prints it before doing the SKIP operation.

```
PRINTEXT  SKIP=1
```

6) Write out the contents of the text statement at the label CODES and do not translate the data.

```
PRINTEXT  CODES,XLATE=NO
```

PRINTEXT - Display a message on a terminal *(continued)*

Coding Examples

1) The PRINTEXT instruction at label P1 sends an untranslated message to an ASCII terminal indicating that a program has begun processing. The example then uses a set of PRINTEXT instructions to print the title of a report on the system printer.

```
TERMMSG  EQU      *
         ENQT     ASCIITEM
P1       PRINTEXT UNXLATED,XLATE=NO
         DEQT
*
HEADER   EQU      *
         ENQT     $SYSPRTR         GET EXCLUSIVE ACCESS TO PRINTER

         PRINTEXT COMPANY,LINE=3,SPACES=39
         PRINTEXT 'ANNUAL INVENTORY REPORT',SPACES=40,SKIP=2
         PRINTEXT 'SCHEDULE D',SPACES=46,SKIP=1
*
PROCESS  EQU      *
            .
            .
         DC       X'1F1F'          DEFINE LENGTH/COUNT BYTES
UNXLATED DC       X'53434845'
         DC       X'44554C45'
         DC       X'20442050'
         DC       X'524F4345'
         DC       X'5353494E'
*
         DC       X'47204841'
         DC       X'53204245'
         DC       X'47554E'
*
COMPANY  TEXT     ' SMITH & JONES CORPORATION'
ASCIITEM IOCB     ACCA64
```

The message written to the ASCII terminal would be displayed as:

```
               SCHEDULE D PROCESSING HAS BEGUN
```

The sequence of lines issued to the enqueued printer would appear as:

```
                                                   (line 0)
                                                   (line 1)
                                                   (line 2)
            SMITH & JONES CORPORATION              (line 3)
                                                   (line 4)
             ANNUAL INVENTORY REPORT               (line 5)
                   SCHEDULE D                      (line 6)
                                                   (line 7)
                                                   (line 8)
```

Note that the line numbers at the right are for reference purposes only and are not part of the printed output.

PRINTEXT - Display a message on a terminal *(continued)*

2) This example shows how to print a message using the character spacing capabilities of the 4975 printer. The MOVE instruction at M1 moves the number of bytes in the PRINTEXT message into CNT+1. After index registers #1 and #2 are set to zero, a DO loop moves the first character of the text message into the buffer BUF. The MOVE instruction at label M2 inserts the spacing character (X'27') and the space modifier (X'F5') into the buffer. The ADD instructions update the pointers. The loop continues until it moves the entire text message into the buffer. The spacing character and the space modifier are inserted between each character in the message.

After the loop completes, the message in the buffer is printed. The spacing between characters in the printed message has increased by 50 percent.

```
SPACING PROGRAM   START
START   EQU       *
M1      MOVE      CNT+1,MSG-1,(1,BYTE)    FIND NUMBER OF BYTES IN MESSAGE
        MOVE      #1,0                    INITIALIZE #1
        MOVE      #2,0                    INITIALIZE #2
*
* THE FOLLOWING LOOP INSERTS SPACING CHARACTERS INTO THE DATA STREAM
*
        ENQT      $SYSPRTR                ENQUEUE 4975 PRINTER
        DO        0,TIMES,P1=CNT          DO FOR NUMBER OF MESSAGE BYTES
        MOVE      (BUF,#2),(MSG,#1),(1,BYTE)  MOVE THE MESSAGE CHARACTER
M2      MOVE      (BUF+1,#2),FRACT,(2,BYTE)   INSERT SPACING CHARACTER
*                                             AND SPACE MODIFIER
        ADD       #1,1                    INCREMENT POINTERS
        ADD       #2,3                    INCREMENT POINTERS
        ENDDO
        MOVE      CNT,#2                  GET TOTAL NUMBER OF CHARACTERS
*                                         TO PRINT
        MOVE      BUF-1,CNT+1,(1,BYTE)
        PRINTEXT  BUF,SKIP=1              PRINT THE MESSAGE
        DEQT
        PROGSTOP
*
FRACT   DATA      X'27F5'                 THE SPACING CHARACTER AND
*                                         SPACE MODIFIER
MSG     TEXT      'THIS IS A TEST MESSAGE'
BUF     TEXT      LENGTH=230
        ENDPROG
        END
```

PRINTEXT - Display a message on a terminal *(continued)*

The message, after the spacing operation, appears as follows:

```
T H I S   I S   A   T E S T   M E S S A G E
```

If no additional spacing were added, the message would have been printed as follows:

```
THIS IS A TEST MESSAGE
```

3) When you code a value with the CAPS operand, the system generates an IOR instruction to capitalize the specified data. The example below shows the use of the CAPS operand and how you can achieve the same results by coding an IOR instruction directly in your application program.

With the CAPS operand

```
                  .
         PRINTEXT A,CAPS=5
                  .
                  .
A        TEXT     LENGTH=5
```

Without the CAPS operand

```
                  .
         IOR      A,X'40',(5,BYTES)
         PRINTEXT A
                  .
                  .
A        TEXT     LENGTH=5
```

PRINTEXT - Display a message on a terminal *(continued)*

4) The following example shows how you can use the PRINTEXT instruction to highlight characters in printed output.

```
SAMPLE   PROGRAM   START
START    EQU       *
         ENQT      $SYSPRTR
         PRINTEXT 'THIS IS AN EXAMPLE SHOWING',MODE=LINE
         PRINTEXT  'HIGHLIGHTING OF CHARACTERS',MODE=LINE
         TERMCTRL  DISPLAY
         PRINTEXT  'HIGHLIGHTING OF CHARACTERS',MODE=LINE,
               SPACES=27
         TERMCTRL  DISPLAY
         PRINTEXT  'ON THE PRINTER',MODE=LINE,SPACES=54
         PROGSTOP
         ENDPROG
         END
```

The highlighted characters appear in bold in the sample below:

THIS IS AN EXAMPLE SHOWING **HIGHLIGHTING OF CHARACTERS** ON THE PRINTER

PRINTEXT - Display a message on a terminal *(continued)*

Request Special Terminal Function (4975-01A)

To request special terminal control function on the 4975-01A ASCII Printer, it is necessary to use the DATA STREAM. The data stream provides terminal control capabilities for the 4975-01A ASCII Printer similar to those provided by the TERMCTRL statement. Unlike the TERMCTRL statement, however, the data stream requires that you code terminal control statements called 'code extension sequences.' These sequences of hexadecimal control characters provide print control function. The printer interprets these characters and prints text accordingly.

This section contains some of the basic sequences required in a data stream on the 4975-01A ASCII Printer. For more information on code extension sequences used with the 4975-01A ASCII printer, refer to the*IBM 4975 Printer Model 01A (7 Bit Code) Description*, GA34-1595.

Do not confuse the 4975-01A ASCII printer with other 4975 printers. The 4975-01A ASCII Printer uses the International Standards Organization Standard 7-Bit Coded Character Set for Information Processing Interchange (ISO-7). Other 4975 printers may not use this character set. The 4975 printer device uses TERMCTRL statements, not the data stream. See "4975 Printer" on page LR-459 for information about TERMCTRL statements for that model printer.

Although most existing programs will generate output on the 4975-01A ASCII Printer, it will ignore TERMCTRL statements.

Code Extension Sequences

Code extension sequences inform the 4975-01A ASCII printer how to interpret data that will follow. You send such sequences from the system to the printer. Among sequences your printer interprets is one which indicates the type of unit spacing. That is the Positioning Unit Mode (PUM) sequence. There are two choices for unit spacing possible. One produces lines and characters per inch. The other makes it possible for you to space units of text precisely within a fraction of an inch called a decipoint. A decipoint is one tenth of a point. A point is 1/12 of a pica. A pica is 1/6 of an inch. There are 720 decipoints in one inch. The two Positioning Unit Modes (PUM) are called:

- Lines and Characters PUM
- Decipoint PUM

To Set Lines and Characters Positioning Unit Mode (PUM)

The PUM code is necessary because spacing increments can be interpreted by the 4975-01A Printer as either lines and characters *or* decipoints. This code makes the distinction.

PRINTEXT - Display a message on a terminal *(continued)*

The 4975-01A ASCII Printer prints text in lines and characters PUM when you code the stream of hexadecimal characters, 1B5B31316C. Since lines and characters per inch is the system default, however, it is not always necessary to include this PUM code. Unless decipoint PUM was previously requested, parameters will automatically be interpreted as lines and characters per inch. Therefore, only your intention to *reset* spacing on the 4975-01A Printer to lines and characters from decipoints is necessary. The meaning of each portion of this code follows.

Byte	Hex	Field
0	1B	Control Sequence Introducer
1	5B	Control Sequence Introducer
2	31	Numeric Parameter for PUM
3	31	Numeric Parameter for PUM
4	6C	Final Character

This sequence causes interpretation of all subsequent numeric parameters (np) in the formatting operations that will follow as units of lines and characters. If the last positioning unit request made of your 4975-01A ASCII Printer was decipoint positioning, include the code 1B5B31316C in your data stream before indicating actual lines and characters spacing increments.

To Set Spacing Increment (SPI)

In order to set spacing increments in lines and or characters or decipoints on the 4975-01A ASCII Printer, include SPI code in the data stream after either PUM code. The SPI code used for indicating lines and characters or decipoints is "1B5Bnp3Bnp2047." The "np" position in the data stream is reserved for numeric parameter coding. The first numeric parameter indicates vertical spacing or lines per inch. Use the second indicates horizontal positioning or characters per inch.

Whether indicating lines and characters or decipoint positioning, numeric parameters in a data stream are simply code equivalents for decipoint spacing values. Numeric parameter values for *each digit* of a decipoint value range from 30 to 39 for 0 to 9 respectively. For example, the np value 35 equals 5 decipoints. The np value 313230 equals 120 decipoints or 12 points. Request the number of lines and characters per inch in the data stream by using the coded equivalent value for the associated numerical parameter.

Decipoint values allowed in a data stream range from 1 to 120. Numerical parameter equivalents range from 31 to 313230. When specifying lines and characters per inch, it is helpful to regard decipoint values as points. For example, 12 decipoints are equal to 12-point type spacing.

Note from the following table that a request for 12-point vertical type spacing results in 6 lines per inch. A request for 9-point vertical type spacing allows 8 lines per inch. Character spacing can also be more easilly understood in points. Horizontal spacing of 7.2 points results in 10 characters per inch. A smaller spacing increment, 4.8 points, allows more characters per inch, 15. There are no vertical nor horizontal lines and characters per inch spacing options available on the 4975-01A Ascii Printer in lines and characters PUM besides these.

PRINTEXT - Display a message on a terminal *(continued)*

The following table illustrates the meaning of code valid in the lines and characters positioning unit mode.

Numeric Parameter	Coded Equivalent	Inch Equivalent
120	313230	6 lines per inch*
90	3930	8 lines per inch
72	3732	10 characters per inch*
48	3438	15 characters per inch

The default number of lines per inch for the ASCII printer is 6. The default number of characters per inch is 10. Specific coding is not required to indicate defaults for lines and characters per inch. They may, however, be coded by numeric parameter equivalents.

If you wish to use any of these parameters code in hexadecimal:

Coded SPI Parameter	Inch Equivalent
1B5B39303B34382047	8 lpi, 15 cpi
1B5B*3B34382047	6 lpi, 15 cpi
1B5B39303B*2047	8 lpi, 10 cpi
1B5B*3B*2047	6 lpi, 10 cpi
1B5B3132303B37322047	6 lpi, 10 cpi

Notes:

1. Asterisks in the tables above indicate that the printer will use the default values. Do not code asterisks in a data stream.
2. Abbreviations "cpi." and "lpi." represent characters and lines per inch respectively.

The meanings of contents of this code are:

Byte	Hex	Field
0	1B	Control Sequence Introducer
1	5B	Control Sequence Introducer
+n	30-39	Numeric Parameter (vertical)
+1	3B	Separator
+n	30-39	Numeric Parameter (horizontal)
+1	20	Intermediate Character
+1	47	Final Character

In this table +n refers to whatever number happens to be the coded equivalent for the numeric parameter you are requesting. It can be four or six digits.

PRINTEXT - Display a message on a terminal *(continued)*

To Set Decipoint PUM

If you want to space text more precisely than lines and characters PUM will allow, consider using decipoint parameters. Issue decipoint PUM code 1B5B313168 in your data stream before introducing specific decipoint horizontal and vertical spacing numeric parameters. SPI code following this PUM code allows data to be positioned in any increment of decipoints. The meaning of each portion of this code follows.

Byte	Hex	Field
0	1B	Control Sequence Introducer
1	5B	Control Sequence Introducer
2	31	Numeric Parameter for PUM
3	31	Numeric Parameter for PUM
4	68	Final Character

This sequence causes interpretation of all subsequent numeric parameters (np) in the following formatting operations as units of decipoints. When submitting information in numerical parameters for interpretation as decipoints, consider each standard numerical parameter unit a decipoint. The following table indicates equivalent (np) code for several decipoint values.

Decipoint Value	Coded (np) Equivalent
120	313230
110	313130
90	3930
80	3830
70	3730
30	3330

To Reset to Initial State (RIS)

This sequence, 1B63, resets the printer to its initial state. The initial state is the printer's state after turned on. This sequence may replace coding for printer defaults.

Byte	Hex	Field
0	1B	Escape Character
1	63	Final Character

Data Stream Example

The following program example demonstrates how to change print density on the 4975-01A ASCII Printer.

Once enqueued, the printer prints text in lines and characters per inch PUM, the default positioning unit mode. Lines and characters will automatically print with a density of 6 lines and 10 characters per inch. The ASCII printer retains any print density information you specify until you request new values by numeric parameter specification or the RIS sequence.

The XLATE= NO operand used in this example sends our message to the device without translation. Results of the program follow the example.

PRINTEXT - Display a message on a terminal *(continued)*

```
PGM       PROGRAM      START
*
START     EQU         *
*
          ENQT         ASCIPRNT      ENQT ON THE PRINTER
          PRINTEXT    'THIS IS 6 LINES/INCH, 10 CHARACTERS/INCH (DEFAULT)'
          PRINTEXT     SKIP=1
          PRINTEXT    'THIS IS 6 LINES/INCH, 10 CHARACTERS/INCH (DEFAULT)'
*
          PRINTEXT    P815,XLATE=NO  CHANGE PRINT DENSITY TO 8 LPI 15 CPI
*
          PRINTEXT    'THIS IS 8 LINES/INCH, 15 CHARACTERS/INCH',SKIP=1
          PRINTEXT    'THIS IS 8 LINES/INCH, 15 CHARACTERS/INCH',SKIP=1
*
          PRINTEXT    P615,XLATE=NO  CHANGE PRINT DENSITY TO 8 LPI 15 CPI
          PRINTEXT    'THIS IS 6 LINES/INCH, 15 CHARACTERS/INCH',SKIP=1
          PRINTEXT    'THIS IS 6 LINES/INCH, 15 CHARACTERS/INCH',SKIP=1
*
          DEQT                       DEQT THE PRINTER
          PROGSTOP
*
ASCIPRNT     IOCB      $SYSPRT2      IOCB FOR THE 4975-01A
*
             DC        X'0909'       DATA TO DEFINE TEXT STRING LENGTH
P815         DC        X'1B5B'       BEGINNING SEQUENCE
             DC        X'3930'       SPECIFIES 8 LPI
             DC        X'3B'         SEPARATOR
             DC        X'3438'       SPECIFIES 15 CPI
             DC        X'2047'       ENDING SEQUENCE
*
             ALIGN     WORD          ALIGN DATA STREAM
*
             DC        X'0707'       DATA TO DEFINE TEXT STRING LENGTH
*
P615         DC        X'1B5B'       BEGINNING SEQUENCE
*                                    NO PARAMETER, MEANS 6 LPI (DEFAULT)
             DC        X'3B'         SEPARATOR
             DC        X'3438'       SPECIFIES 15 CPI
             DC        X'2047'       ENDING SEQUENCE
*
          ENDPROG
          END
```

The following output results from the preceding program example:

```
THIS IS 6 LINES/INCH, 10 CHARACTERS/INCH (DEFAULT)
THIS IS 6 LINES/INCH, 10 CHARACTERS/INCH (DEFAULT)
THIS IS 8 LINES/INCH, 15 CHARACTERS/INCH
THIS IS 8 LINES/INCH, 15 CHARACTERS/INCH
THIS IS 6 LINES/INCH, 15 CHARACTERS/INCH
THIS IS 6 LINES/INCH, 15 CHARACTERS/INCH
```

PRINTEXT - Display a message on a terminal *(continued)*

Terminal I/O Return Codes

The terminal I/O return codes are all listed here and following the READTEXT instruction. A complete list of all return codes can also be found in the *Messages and Codes*. You must select the group of codes that represents the particular device type you are using. A list of the terminal I/O return code groups follows:

- General Terminal I/O
- Virtual Terminal
- ACCA Devices
- Interprocessor Communication
- General Purpose Interface Bus
- Series/1-to-Series/1 Adapter.

PRINTEXT - Display a message on a terminal *(continued)*

General Terminal I/O Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-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.
8	Printer power has been switched off and switched back on or a power failure has occurred.
>10	A code greater than 10 can indicate 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 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

Notes:

1. If the return code is for devices supported by IOS2741 (2741, PROC) and a code greater than 128 is returned, subtract 128; the result then contains status word 1 of the ACCA. Refer to the *IBM Series/1 Communications Features Description*, GA34-0028 for determination of the special error condition.

2. If your program receives a return code of 5 while attempting to do a PRINTEXT operation on a 4975 printer, the program should retry the operation a maximum of three times.

PRINTEXT - Display a message on a terminal *(continued)*

Virtual Terminal Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Transmit Condition	Receive Condition
X'8Fnn'	Not applicable.	LINE=nn received.
X'8Enn'	Not applicable.	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.

A further description of each of the virtual terminal return codes follows:

LINE=nn (X'8Fnn'): Returned for a READTEXT or GETVALUE instruction if the other program issued an instruction with a LINE= operand. This operand tells the system to do an I/O operation on a certain line of the page or screen. The return code allows the receiving program to reproduce on an actual terminal the output format intended by the sending program.

SKIP=nn (X'8Enn'): The other program issued an instruction with a SKIP= operand. This operand tells the system to skip several lines before doing an I/O operation.

Line Received (-2): Indicates that an instruction (usually READTEXT or GETVALUE) has sent information but has not issued a carriage return to move the cursor to the next line. The information is usually a prompt message.

New Line Received (-1): Indicates transmission of a carriage return at the end of the data. The cursor is moved to a new line. This return code and the Line Received return code help programs to preserve the original format of the data they are transmitting.

Not attached (1): A virtual terminal does not or cannot refer to another virtual terminal.

Disconnect (5): The other virtual terminal program ended because of a PROGSTOP or an operator command.

Break (8): Indicates that both virtual terminal programs are attempting to do the same type of operation. When one program is reading (READTEXT or GETVALUE), the return code means the other program has stopped sending and is waiting for input. When one program is writing (PRINTEXT or PRINTNUM), the return code means the other program is also attempting to write.

If you defined only one virtual terminal with SYNC=YES, then that task always receives the break code. If you defined both virtual terminals with SYNC=YES, then the task that last attempted the operation receives the break code.

PRINTEXT - Display a message on a terminal *(continued)*

ACCA Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful completion.
1-08	Return code for last operation placed in information status byte (ISB). Refer to the hardware description manual for status on the device you are using.
11	Write operation (I/O complete).
12	Read operation (I/O complete).
14,15	Condition code +1 after I/O start or condition code after I/O complete.

Interprocessor Communication Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

CODTYPE=	Return Code	Condition
EBCDIC	FDFF	End of transmission (EOT).
EBCDIC	FEFF	End of record (NL).
EBCDIC	FCFF	End of subrecord (EOSR).
EBCD/CRSP	1F	End of transmission (EOT).
EBCD/CRSP	5B	End of record (NL).
EBCD/CRSP	(none)	End of subrecord (EOSR).

General Purpose Interface Bus Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful completion.
1	Device not attached.
2	busy condition.
3	busy after reset.
4	command reject.
6	Interface data check.
256 + ISB	Read exception.
512 + ISB	Write exception.
1024	Attention received during an operation (may be combined with an exception condition).

PRINTEXT - Display a message on a terminal *(continued)*

Series/1-To-Series/1 Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful.
1	Device not attached.
2	System error (busy condition).
3	System error (busy after reset).
4	System (command reject).
5	Device not ready (not reported for S/1 - S/1).
6	Interface data check.
7	Overrun recieved (not reported for S/1 - S/1).
138, 154	An error has occurred that can only be determined by displaying the device cycle steal status word with the TERMCTRL STATUS function and checking the bits to determine the cause of the error.
1002	Other system not active.
1004	Checksum error detected.
1006	Invalid operation code or sequence.
1008	Timeout on data transfer.
1010	TERMCTRL ABORT issued by responding processor.
1012	Device reset (TERMCTRL RESET) issued by the other processor.
1014	Microcode load to attachement failed during IPL.
1016	Invalid or unsolicited interrupt occurred.
1050	TERMCTRL ABORT issued and no operation pending.
1052	TERMCTRL IPL attempted by slave processor.
1054	Invalid data length.

PRINTIME - Display the time on a terminal

The PRINTIME instruction prints the time of day on the currently enqueued terminal. The system prints the time in the form HH:MM:SS (hours, minutes, seconds), according to a 24-hour clock. You set the 24-hour clock with the $T command.

Note: To use the PRINTIME instruction, you must have installed timer hardware and included timer support in the system during system generation. A program check will occur if you try to use this instruction without the proper hardware or software support.

The supervisor places a return code in the first word of the task control block (taskname) whenever a PRINTIME instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

Syntax:

```
label         PRINTIME

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

Operand	*Description*
none	none

3101 Display Considerations

If you use a 3101 in block mode, the current TERMCTRL command in effect will control the output. For details on the TERMCTRL SET,ATTR and SET,STREAM operands, see the discussion under "TERMCTRL - Request special terminal functions" on page LR-446.

PRINTIME - Display the time on a terminal *(continued)*

Coding Example

The following coding example prints a message on the system printer, followed by the current time of day.

```
  .
  .
  .
ENQT     $SYSPRTR
PRINTEXT '@ THE TIME IS '
PRINTIME
DEQT
.
.
```

If, for example, the PRINTIME instruction executes at 10 minutes and 13 seconds past 2 o'clock in the afternoon, the instruction prints the following message on the system printer:

```
THE TIME IS  14:10:13
```

PRINTNUM - Display a number on a terminal

The PRINTNUM instruction displays or prints a floating-point value or one or more integer values on a terminal in the format that you specify. The output can appear in decimal or hexadecimal form.

If the PRINTNUM output is too large for the system buffer, the system first fills the buffer, prints that data, and then stores the excess data in the buffer area. The next I/O operation forces the excess data to be printed or displayed before any other output.

The supervisor places a return code in the first word of the task control block (taskname) whenever a PRINTNUM instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). However, if an I/O error occurs during this instruction, terminal I/O will not pass control to any terminal error routine. The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

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
```

Operand	***Description***
loc	The label of the first value to be printed or displayed. Successive values are taken from successive words or doublewords.
count	The number of values to be printed or displayed. You can substitute a precision for the count, in which case the count defaults to 1. The valid precisions are WORD (the default) and DWORD (doubleword). You can also express the count in the form: (count,precision).
nline	The number of values to be printed or displayed on each line.
nspace	The number of spaces left between values. Code the nline operand before coding this operand.

PRINTNUM - Display a number on a terminal *(continued)*

MODE= HEX, for hexadecimal output.

DEC, the default, for decimal output.

FORMAT= (w,d,t) The format of the value to be printed or displayed. If you code this operand, the system ignores the count, nline, nspace, and MODE= operands. The format is as follows:

w An integer value equal to the width of the data field to be printed or displayed. If the data contains a decimal point or sign character (+ or -), include it in the count.

d The number of 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. Code I for integer data, F for floating-point data (XXXX.XXX), or E for floating-point data in E notation. See the value operand under the DATA/DC statement for a description of E notation format.

Note: You can use the floating-point format for data even if you do not have floating-point hardware installed in your system. Floating-point hardware is required, however, to do floating-point arithmetic.

The first FORMAT operand to execute generates a work area which all subsequent FORMAT operands also use. The generated work area is nonreentrant in a multitasking environment, and all tasks must use the ENQ and DEQ instructions to acquire serial access to it.

TYPE= The type of variable that contains the data you want to print or display. Code this operand only when you code 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 system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

PRINTNUM - Display a number on a terminal *(continued)*

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

PROTECT= Code PROTECT=YES to write protected characters to a device for which this feature is supported.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

PRINTNUM - Display a number on a terminal *(continued)*

3101 Display Considerations

If you use a 3101 in block mode, the most recent TERMCTRL command will control the output. For details on the discussion under TERMCTRL SET,ATTR and SET,STREAM operands, see "TERMCTRL - Request special terminal functions" on page LR-446.

Syntax Examples

1) Print the first value in A in integer format.

```
PRINTNUM  A
```

2) Print the first 10 values in BUF1 in integer format.

```
PRINTNUM  BUF1,10
```

3) Print the first value in AX in hexadecimal form.

```
PRINTNUM  AX,MODE=HEX
```

4) Print the first 10 values in BUF2, put five values on each line, and print three spaces between each value.

```
PRINTNUM  BUF2,10,5,3
```

5) Print the first 10 doublewords of BZ in hexadecimal form.

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

6) Print 8 numbers, four in a line, with 5 spaces between the numbers.

```
PRINTNUM  NUMBERS,8,4,5
```

PRINTNUM - Display a number on a terminal *(continued)*

Coding Example

The following example uses the PRINTNUM instruction to display a floating-point value and an integer value on a terminal. The system displays the values on the terminal you use to load the program.

The program first asks you to enter a floating-point number. The GETVALUE instruction places the number you enter in FLCOUNT. At label LOOP1, the program begins a loop that adds the floating-point number in FLCOUNT to the contents of FLSUM ten times. The second GETVALUE instruction asks you to enter an integer. It places the value you enter in INTCOUNT. The DO loop at label LOOP2 adds the integer value in INTCOUNT to the contents of INTSUM ten times.

The PRINTNUM instruction at PRINT1 displays the contents of FLSUM in floating-point format. The PRINTNUM instruction at PRINT2 displays the contents of INTSUM in integer format.

```
PROG1    PROGRAM  START,FLOAT=YES
START    EQU      *
         GETVALUE FLCOUNT,'ENTER FLOATING POINT NUMBER: '           X
               TYPE=F,FORMAT=(4,3,F)
LOOP1    DO       10,TIMES
           FADD     FLSUM,FLCOUNT
         ENDDO
         GETVALUE INTCOUNT,'ENTER INTEGER NUMBER: '
LOOP2    DO       10,TIMES
           ADD      INTSUM,INTCOUNT
         ENDDO
         PRINTEXT '@FLOATING POINT RESULT= '
PRINT1   PRINTNUM FLSUM,FORMAT=(5,2,F),TYPE=F
         PRINTEXT '@INTEGER RESULT= '
PRINT2   PRINTNUM INTSUM
         PROGSTOP
INTCOUNT DATA     F'0'
FLCOUNT  DATA     E'0.000'
FLSUM    DATA     E'00.00'
INTSUM   DATA     F'0'
         ENDPROG
         END
```

PROGRAM - Define your program

The PROGRAM statement defines the primary task of your program and the resources your program uses. PROGRAM is the first statement you code in every application program assembled using $EDXASM or $S1ASM.

You can only omit the PROGRAM statement when you are compiling a subprogram under $EDXASM. (See the MAIN operand for a definition of a subprogram.) When program assembly is to be done by the Host or Series/1 macro assemblers, you must code a PROGRAM statement even for subprograms.

Syntax:

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

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

Operand	***Description***
taskname	The label you assign to the primary task of the program. The system generates a control block for each task in the program. This control block is known as a task control block (TCB). The system generates the TCB when it encounters an ENDPROG statement. The label of the primary task's TCB is the label you specify with this operand. The supervisor uses the TCB to store instruction return codes. By referring to the TCB (the taskname) in your program, you can determine if an operation completed successfully.
start	The label of the first instruction to be executed in your program. The instruction must be on a fullword boundary.
priority	The priority of the program's primary task. The system uses priorities to establish the order in which it executes tasks. Tasks with high priorities are executed before tasks with low priorities. The range is from 1 (highest priority) to 510 (lowest priority). Priorities 1-255 imply foreground operation and are executed on hardware interrupt level 2. Priorities 256-510 imply background operation and are executed on interrupt level 3.
EVENT=	The label of the event to be posted when the system detaches the primary task. Use this operand only if another task will issue a WAIT for this event. Do not

PROGRAM - Define your program *(continued)*

code an event control block (ECB) with this label because the system generates the ECB for you. An error message appears at the end of the program compiler listing if this event is defined more than once.

DS= Names of 1-9 disk, diskette, or tape data sets to be used by this program. Each name is composed of 1-8 alphameric characters, the first of which must be alphabetic. Only one tape data set for each tape volume can be specified.

If your program retrieves formatted messages from a disk or diskette data set, you must specify the data set name with this operand. The COMP statement in your program provides the location of the message by referring to the data set list on the PROGRAM statement.

The system automatically generates one data set control block (DSCB) in the program header for each data set you specify on the DS operand of the PROGRAM statement. The system gives each DSCB the name DSx, where x is the position of a data set in the list of data sets you code on this operand. The DSCB named DS1, for example, corresponds to the first data set in the DS= list. You can refer to fields within a DSCB with the expression DSx+name, where "name" is a label defined in the DSCB equate table, DSCBEQU. You must include the following statement in your source program when you refer to DSCB fields:

```
COPY    DSCBEQU
```

If the special characters ## are found in a program header in place of a volume name, the name of the volume from where the main program was loaded is substituted for the ## characters. This allows data sets specified in the program header to reside on the same volume as the main program.

All tape data sets are of the form (DSN,VOLUME). The specification of tape data sets is dependent on 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 such as 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 is enclosed in a second set of parentheses. For example:

PROGRAM - Define your program *(continued)*

```
...,DS=((MYDS,MYVOL))
```

refers to the data set MYDS on volume MYVOL. In the following example:

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

DS= refers to the data set ACTPAY on volume EDX001 and to DSDATA2 on volume EDX003.

If you do not specify a volume, the default is the IPL volume. When one data set is used and it is in the IPL volume, no parentheses are required. For example:

```
...,DS=CUSTFIL
```

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

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

Four special data set names are recognized: ??, $$EDXLIB, and $$ or $$EDXVOL. A data set control block (DSCB) is created just as for any other data set name. However, special 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, you will receive the following prompt message:

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

If the specified sequence is of the form,

```
...DS=??
```

you will receive the prompt message,

PROGRAM - Define your program *(continued)*

DSn(NAME,VOLUME):

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

If $$EDXLIB or $$ is used as a data set name with disks, 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. Symbol $$ also can be used to reserve a DSCB in the program header so that it can be filled in and opened (using DSOPEN) after execution begins.

With diskettes, $$EDXVOL only references records on cylinder 0. If a single-density diskette is used, $$EDXVOL references records 1 to 26. With a double-density diskette, $$EDXVOL references records 1 to 39. Symbol $$ and $$EDXLIB reference diskette records beginning with cylinder 1.

PARM= A word count specifying the length of a parameter list to be passed to this program at load time. Each word in the list can be referred to by the name $PARMx, where "x" is the position or number of the word in the list beginning with 1. The maximum length of this list is 762 words less 33 for each data set name you specified in the DS operand and each overlay program name you specified in the PGMS operand.

This operand 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 on its contents. If a program is loaded using $L and it has a PARM specification, the parameters will be initialized to zero.

PGMS= The names of 1-9 programs that can be loaded as overlay programs during the 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 that apply to the DS operand apply to this operand.

The system reserves space within this program for the largest of the overlay programs identified in this list, thus ensuring that space will be available for the overlays when the program is executed.

You invoke program overlays with the LOAD instruction. Only one overlay program can execute at a time because each uses the same storage area. See the description of the LOAD instruction for additional information.

Note: You can only code this operand in a main program and not on the PROGRAM statement of an overlay program. In addition, you cannot code this operand for tape data sets.

The system automatically generates one DSCB in the program header for each overlay program you specify on the PGMS operand of the PROGRAM

PROGRAM - Define your program *(continued)*

statement. The system gives each DSCB the name PGMx, where "x" is the position of an overlay in the list of overlay programs you code on this operand. The DSCB named PGM1, for example, corresponds to the first data set in the PGMS= list. You can refer to fields within a DSCB with the expression PGMx+name, where "name" is a label defined in the DSCB equate table, DSCBEQU. You must include the following statement in your source program when you refer to DSCB fields:

```
        COPY    DSCBEQU
```

If the special characters ## are found in a program header in place of a volume name, the name of the volume from where the main program was loaded is substituted for the ## characters. This allows overlays specified in the program header to reside on the same volume as the main program.

TERMERR= The label of the routine to receive control if an unrecoverable terminal I/O error occurs.

If such an error occurs, the first word of the task control block (TCB) contains the return code indicating the error. The second word of the TCB contains the address of the instruction that was executing when the error occurred.

If TERMERR is not coded, the return code is available in the task code word. Use of TERMERR, however, is the recommmended method for detecting errors because the task code word is subject to modification by numerous system functions. It may not, therefore, always reflect the true status of terminal I/O operations.

FLOAT= YES, if the primary task uses floating-point instructions.

NO (the default), if the primary task does not use floating-point instructions.

MAIN= YES, if this program contains the primary task.

NO, if this program does not contain the primary task. 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. When a subprogram is to be assembled by $EDXASM, the PROGRAM statement may be omitted entirely.

You link-edit program modules with the $EDXLINK utility. For information on the $EDXLINK utility, refer to the *Operator Commands and Utilities Reference*

Note: Subprograms must not contain TASK, ENDTASK, IODEF, 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

PROGRAM - Define your program *(continued)*

MAIN=NO is specified, then none of the other operands of the PROGRAM statement are meaningful.

ERRXIT= The label of a 3-word area that 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 *Event Driven Executive Language Programming Guide* for additional information on the use of task error exit routines.

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 allows you to take whatever action is appropriate.

The format of the task error exit area is:

WORD 1 The count of the number of parameter words which follow (always F'2').

WORD 2 The address of your 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.

A default task error exit routine is available to aid in problem diagnosis and correction. (Refer to the *Event Driven Executive Language Programming Guide* for a detailed description of this routine and how to use it in your application program.)

STORAGE= The number of bytes of additional storage the system should allocate for this program when the program 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 referred to by your program during execution. Do not use this operand if you are loading the program as an overlay.

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.

PROGRAM - Define your program *(continued)*

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.

Syntax Examples

1) TASK1 is the label of the primary task and the label of the first executable instruction is START. The priority of TASK1 is the default priority, 150.

```
TASK1     PROGRAM   START
```

2) The primary task, TASK2, has a priority of 300 and starts at the label BEGIN. The program uses floating-point instructions.

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

3) The primary task, TASK3, starts at GOPROG. One data set, NAME1, is defined and is located in the volume from which the main program will be loaded. Disk I/O instructions in the program refer to NAME1 by the symbolic name DS1.

```
TASK3     PROGRAM   GOPROG,DS=((NAME1,##))
```

4) The primary task, TASK4, starts at START4 and uses one tape data set. The 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. You write such labels using the INITIALIZE function of the $TAPEUT1 utility.

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

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

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

PROGRAM - Define your program *(continued)*

6) The primary task, TASK6, starts at START6. Two data sets are defined. The name of the first data set will be specified at program load time. The second data set 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.

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

7) The primary task, TASK7, starts at START7 and uses 4 data sets. MYDS1 is a disk or diskette data set on the IPL volume. MYDS2 is a tape data set on standard labeled tape number 100001. The program prompts the operator for the last two data sets. The prompt for the third data set appears as OUTPUT(NAME,VOLUME); the prompt for the fourth data set appears as DS4(NAME,VOLUME). The operator can specify the third and fourth data sets as disk, diskette, or tape data sets.

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

PROGSTOP - Stop program execution

The PROGSTOP instruction ends program execution and releases the storage allocated to the program. You can have more than one PROGSTOP instruction in a program. You are responsible for ensuring that any secondary tasks in a program are inactive before a PROGSTOP statement is executed by the primary task. 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 issuing a PROGSTOP. The following instructions can generate asynchronous events: READ, WRITE, STIMER, LOAD, ENQ, and ENQT. Also, if another program can post your program, you must wait for the post or prohibit the other program from posting before the PROGSTOP executes.

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

PROGSTOP will do a DEQT of the terminal currently in use by the program.

When coding the PROGSTOP instruction, you can include a comment which will appear with the instruction on your compiler listing. If you include a comment, you must specify at least one operand with the instruction. The comment must be separated from the operand field by one or more blanks and it may not contain commas.

Syntax:

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

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

Operand	***Description***
code	The posting code to be inserted in the EVENT named in the associated LOAD instruction. The PROGSTOP instruction causes the system to post the ECB for this event, following the post code rules. This operand must be a self-defining term other than 0.

PROGSTOP - Stop program execution *(continued)*

LOGMSG= Code either YES or NO to show whether a "PROGRAM ENDED" message is to be displayed on the terminal being used by this program.

Notes:

1. Programs loaded by the virtual terminal facility do not recognize the LOGMSG operand. Therefore, if a program is loaded by a virtual terminal, the "program ended" message is never displayed.

2. If you coded LOGMSG=YES, but another task has control of the terminal when your program ends, the system does not display the "program ended" message.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

PUTEDIT - Collect and store data from a program

The PUTEDIT instruction obtains data from variables within a program, converts the data to a character string, and either stores the data in a storage area or sends it to a terminal.

PUTEDIT uses the specified FORMAT statement and the data list to convert and move elements one by one into a storage area.

When you use the PUTEDIT instruction in your program, you must link-edit your program using the "autocall" option of $EDXLINK. Refer to the *Event Driven Executive Language Programming Guide* for information on how to link-edit programs.

The supervisor places a return code in the first word of the task control block (taskname) whenever a PUTEDIT instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2).

The system will not pass control to a terminal error routine if an I/O error occurs while this instruction is executing. The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

Syntax:

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

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

Operand	*Description*
format	The label 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 label of a TEXT statement defining a storage area for character data. If data is moved to a terminal, this area stores the data (as an EBCDIC character string) after it is converted from the variables and before it is sent to the terminal. **Note:** The TEXT statement must be large enough to contain all the EBCDIC characters generated by this instruction.

PUTEDIT - Collect and store data from a program *(continued)*

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 label 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 variable to be converted

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

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

format list A FORMAT list. If you want 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 instruction operands which are to be referred to by PUTEDIT instructions.

This operand is not allowed if the program is assembled with $EDXASM.

PUTEDIT - Collect and store data from a program *(continued)*

ERROR= The label of the first instruction of the routine to receive control if an error occurs during the PUTEDIT operation. The system returns a return code to the task even if you do not code this operand.

Errors that might cause the system to invoke the error routine are:

- Use of incorrect format list
- Not enough space in text buffer to satisfy the data list.

ACTION= IO (the default), causes a PRINTEXT to be executed following the data conversion. If output is being directed to a 3101 in block mode, refer to the "PRINTEXT - Display a message on a terminal" on page LR-324 for special considerations.

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 system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

PUTEDIT - Collect and store data from a program *(continued)*

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

PROTECT= YES, to write protected characters to a static screen device that supports this feature, such as an IBM 4978, 4979, 4980, or 3101 in block mode. Protected characters cannot be typed over nor displayed.

NO (the default), to inhibit writing protected characters to a static screen device.

MODE= Tells the system whether you want imbedded @ characters interpreted as new-line designators. Code LINE if the text includes imbedded @ characters which are not to be interpreted as new-line designators.

For 4978, 4979, and 4980 screens accessed in static mode, the coding of MODE=LINE and the spaces parameter (SPACES=) causes the system to skip over protected fields as the data is transferred to the screen. (Protected fields do not contribute to the count.)

For a 3101 in block mode with a static screen, the system overwrites protected fields.

Do not code this parameter if you want the system to interpret @ characters as new line designators.

3101 Display Considerations

When using a 3101 in block mode, the output will be controlled by the most recent TERMCTRL command. For details on the discussion under TERMCTRL SET,ATTR and SET,STREAM operands, see “TERMCTRL - Request special terminal functions” on page LR-446.

PUTEDIT - Collect and store data from a program *(continued)*

Syntax Example

This example converts the integer A into the first four positions of TEXT1 followed by a carriage return command. Then, the next six positions will contain the variable B followed by two spaces. The literal 'DATA=' then follows with the extended-precision variable C converted into the last 10 positions.

```
         PUTEDIT   FM,TEXT1,(A,(B,F),(C,L))
            .
            .
            .
TEXT1    TEXT      LENGTH=28
FM       FORMAT    (I4/F6.2,2X,'DATA=',E10.4)
```

Coding Example

The program issues a PRINTEXT instruction that requests the model year and serial numbers for the automobile of interest. The first GETEDIT reads the two requested numbers into a TEXT statement labeled TEXT1.

The GETEDIT instruction searches the TEXT1 data and converts the first entry to a single-precision variable called LIST1. The second entry is converted to a double-precision variable called LIST2. The first PUTEDIT instruction, using the FORMAT statement labeled PE1FMT, converts LIST1 and LIST2 back to EBCDIC and displays these values on the screen or printer. The PUTEDIT and FORMAT statements determine the layout of the data as it is displayed.

The GETEDIT instruction after label GE2 takes the data already entered into TEXT1 with the preceding READTEXT and converts it into the two binary variables called LIST1 (single-precision) and LIST2 (double-precision). Because ACTION=STG, a READTEXT must be issued before executing the GETEDIT.

The PUTEDIT instruction at label PE2 converts the two variables back to EBCDIC and places them into the TEXT2 statement as formatted by the PE2FMT FORMAT statement. Again the keyword ACTION=STG prevents the data from being displayed until the following PRINTEXT instruction is executed.

PUTEDIT - Collect and store data from a program *(continued)*

```
GE1       EQU      *
          PRINTEXT '@ENTER MODEL YEAR AND SERIAL NUMBER@'
          GETEDIT  GE1FMT,TEXT1,(LIST1,(LIST2,D)),ACTION=IO,ERROR=ERR1
*
PE1       EQU      *
          ENQT     $SYSPRTR
          PUTEDIT  PE1FMT,TEXT2,(LIST1,(LIST2,D)),ACTION=IO
          DEQT
*
GE2       EQU      *
          READTEXT TEXT1,'@ENTER YOUR DEPT. AND SYSTEM ID NUMBER@'
*
          GETEDIT  GE2FMT,TEXT1,(LIST1,(LIST2,D)),                    X
                ACTION=STG,ERROR=ERR1
*
PE2       EQU      *
          PUTEDIT  PE2FMT,TEXT2,(LIST1,(LIST2,D)),ACTION=STG
*
          ENQT     $SYSPRTR
          PRINTEXT TEXT2
          DEQT
           .
           .
           .
ERR1      EQU      *
          PRINTEXT '@GETEDIT GE1 HAS FAILED@'
          GOTO     ERROROUT
*
ERR2      EQU      *
          PRINTEXT '@GETEDIT GE2 HAS FAILED@'
          GOTO     ERROROUT
*
ERROROUT   .
           .
GE1FMT    FORMAT   (I4,1X,I8)
PE1FMT    FORMAT   ('MDL. YR. = ',I4,6X,'SER. NO. = ',I8)
GE2FMT    FORMAT   (I3,1X,I6)
PE2FMT    FORMAT   ('DEPT. = ',I3,4X,'SYST. ID. = ',I6)
LIST1     DATA     F'0'
LIST2     DATA     D'0'
TEXT1     TEXT     LENGTH=13
TEXT2     TEXT     LENGTH=42
```

Return Codes

The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
3	Conversion error

QCB - Create a queue control block

The QCB statement generates a 5-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 you as a consequence of naming a QCB in a DEQ instruction. However, it can be used for special purposes such as controlling their location within a program. You must code any necessary QCBs in programs you compile with $EDXASM.

A program can contain a maximum of 25 QCB statements. If more than 25 QCBs are required, you must create them with the DATA statement. For example:

```
QCB1        QCB
```

is equivalent to coding,

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

When coding the QCB statement, you can include a comment which will appear with the statement on your compiler listing. If you include a comment, you must also specify the code operand. The comment must be separated from the operand field by at least one blank and it may not contain commas.

Syntax:

```
label       QCB     code        comment

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

Operand	***Description***
label	The label of the QCB statement. The ENQ and DEQ instructions refer to this label.
code	Initial value of the code field (word 1). If this word is nonzero, the resource this QCB refers to is available for use by a program or task.

QCB - Create a queue control block *(continued)*

Coding Example

The QCB statement labeled SBRTNQCB generates a 5-word queue control block (QCB). The ENQ instruction checks the QCB to see if the subroutine named SUBRTN is being used by another program or task. The initial value of the QCB is 99, indicating that the resource is initially available for use.

```
          ENQ      SBRTNQCB
          CALL     SUBRTN
          DEQ      SBRTNQCB
           .
           .
          SUBROUT SUBRTN
           .
          RETURN
           .
           .
SBRTNQCB  QCB      99
```

QUESTION - Ask operator for input

The QUESTION instruction allows the terminal operator to choose the direction of a conditional branch in a 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.

The supervisor places a return code in the first word of the task control block (taskname) whenever a QUESTION instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

Syntax:

```
 label        QUESTION pmsg,YES=,NO=,SKIP=,LINE=,SPACES=,
                       COMP=,PARMS=(parm1,...,parm8),
                       MSGID=,P1=

Required:    pmsg and either YES= or NO=
Defaults:    If the operator enters a response and you have
             not coded a keyword for that response (YES= or NO=),
             the system executes the next instruction in the program.
             MSGID=NO
Indexable:   pmsg,SKIP,LINE,SPACES
```

Operand	***Description***
pmsg	The prompt message. Code either the label of a TEXT statement or an explicit text message enclosed in single quotes. To retrieve a prompt message from a data set or module containing formatted program messages, code the number of the message you want displayed or printed. You must code a positive integer or a label preceded by a plus sign (+) that is equated to a positive integer. If you retrieve a prompt message from storage, you must also code the COMP= operand. See Appendix E, "Creating, Storing, and Retrieving Program Messages" on page LR-615 for more information.
YES=	The label of the instruction at which execution will continue if the answer is YES.
NO=	The label at which execution will continue if the answer is NO.

QUESTION - Ask operator for input *(continued)*

SKIP= The number of lines to be skipped before the system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

COMP= The label of a COMP statement. You must specify this operand if the QUESTION instruction is retrieving a prompt message from a data set or module containing formatted program messages. The COMP statement provides the

QUESTION - Ask operator for input *(continued)*

location of the message. (See the COMP statement description for more information.)

PARMS= The labels of data areas containing information to be included in a message you are retrieving from a data set or module containing formatted program messages. You can code up to eight labels. If you code more than one label, you must enclose the list in parentheses.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

MSGID= YES, if you want the message number and four-character prefix to be printed at the beginning of the message you are retrieving from a data set or module containing formatted program messages. See the COMP statement operand 'idxx' for a description of the four-character prefix.

NO (the default), to prevent the system from printing or displaying this information at the beginning of the message.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Special Considerations

To use the QUESTION instruction with a static screen, you must create an unprotected input area for the answer to the QUESTION prompt. The QUESTION instruction regards the first nonblank character following the QUESTION prompt as the answer to the prompt message. One or more blanks can precede the answer, but they are not required.

3101 Terminals

If you use a 3101 in block mode, the most recent TERMCTRL SET,ATTR will control the attribute bytes used for the prompt and response.

Neither a TERMCTRL SET,ATTR=BLANK nor SET,STREAM=YES should be in effect when a QUESTION instruction executes.

QUESTION - Ask operator for input *(continued)*

Syntax Examples

1) Ask the operator if he or she wants to start a second routine. If the operator answers "yes", branch to the label ROUTINE2. If the operator answers "no", execute the next instruction.

```
         QUESTION  TEXT3,YES=ROUTINE2         NO = NEXT STATEMENT
            .
            .
ROUTINE2 EQU       *
            .
            .
TEXT3    TEXT      'GO TO SECOND ROUTINE?'
```

2) Ask the operator if he or she wants to do an operation again. If the operator answers "no", branch to the label EXIT. If the operator answers "yes", execute the next instruction.

```
         QUESTION  'DO IT AGAIN?',NO=EXIT   YES = NEXT STATEMENT
            .
            .
            .
EXIT     EQU       *
         PROGSTOP
```

3) Ask the operator if he or she wants to restart an operation. If the operator answers "yes", branch to the label INITIAL. If the operator answers "no", branch to the label END.

```
INITIAL  EQU       *
            .
            .
         QUESTION  'RESTART?',YES=INITIAL,NO=END
            .
            .
END      EQU       *
         PROGSTOP
```

Coding Example

In the following example, the QUESTION instruction displays a prompt message contained in MSGMOD, a storage-resident message area. Because +MSG77 equals 77, the system retrieves message 77 in MSGMOD.

```
         QUESTION   +MSG77,COMP=MSGSTMT,YES=OKAY
            .
OKAY     EQU        *
         PROGSTOP
            .
MSG77    EQU        77
MSGSTMT  COMP       'SRCE',MSGMOD,TYPE=STG
```

QUESTION - Ask operator for input *(continued)*

Message Return Codes

The system issues the following return codes when you retrieve a prompt message from a data set or module containing formatted program messages. The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Message successfully retrieved
301-316	Error while reading message from disk. Subtract 300 from this value to get the actual return code. See the disk return codes following the READ or WRITE instruction for a description of the code.
326	Message number out of range
327	Message parameter not found
328	Instruction does not supply message parameter(s)
329	Invalid parameter position
330	Invalid type of parameter
331	Invalid disk message data set
332	Disk message read error
333	Storage-resident module not found
334	Message parameter output error
335	Disk messages not supported (MINMSG support only)

RDCURSOR - Store static screen cursor position

The RDCURSOR instruction stores the cursor position in a set of data areas you specify. The cursor position is defined as the line number and the number of spaces the cursor is indented from the left margin of the logical screen. RDCURSOR is only valid for terminals with a static screen. For information on defining a static screen see the SCREEN= operand of the IOCB statement or refer to the *Event Driven Executive Language Programming Guide.*

The supervisor places a return code in the first word of the task control block (taskname) whenever a RDCURSOR instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes.*

If you code RDCURSOR after a WAIT KEY instruction for a 3101 in block mode, use a PF key and not the SEND key to end the wait. If you use the SEND key, it positions the cursor at the beginning of the next line and RDCURSOR cannot capture the screen coordinates.

Syntax:

```
label         RDCURSOR    line,indent

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

Operand	***Description***
line	The label of the variable in which the cursor position, relative to the top margin of the logical screen, is to be stored. If the cursor lies outside the line range of the logical screen, then a value of -1 is stored.
indent	The label 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 a value of -1 is stored.

RDCURSOR - Store static screen cursor position *(continued)*

Coding Example

This example defines a terminal with an IOCB statement, then issues an ENQT instruction to that terminal. The terminal name is DISP2. An ERASE instruction clears the screen. The example uses the RDCURSOR instruction to find the cursor position. RDCURSOR puts the relative line position of the logical screen in the the variable labeled LN. It puts the spaces value or column position in the variable labeled COL. Because the exact position of the cursor is known, any terminal I/O issued to this terminal can position the cursor using the LN and COL values as a reference point.

After additional processing, index register #1 is set to a value of 2 with a MOVE instruction. A second RDCURSOR instruction is issued and #1 is used to increase the storage locations by a value of 2 where the new locations are to be stored. This RDCURSOR places the cursor line number and spaces in variables NEXT1 and NEXT2, respectively. NEXT1 and NEXT2 then become the new reference point of the cursor for any additional I/O operations.

```
TUBE      IOCB      DISP2,SCREEN=STATIC   DEFINE THE TERMINAL TO BE
*                                         USED
           .
           .
          ENQT                            GET EXCLUSIVE ACCESS OF
*                                         DISP2
          ERASE     MODE=SCREEN,TYPE=ALL  CLEAR THE SCREEN
*
          RDCURSOR  LN,COL                GET CURSOR POSITION AND PUT
*                                         LINE NUMBER IN LN AND SPACES
*                                         IN COL
           .
           .
          MOVE      #1,2                  SET #1 TO 2
          RDCURSOR  (LN,#1),(COL,#1)      GET CURSOR POSITION AND PUT
*                                         VALUES IN NEXT1 AND NEXT2
*                                         COL

          DEQT                            RELEASE EXCLUSIVE CONTROL OF
*                                         THE TERMINAL
           .
           .
LN        DATA      F'0'
NEXT1     DATA      F'0'
COL       DATA      F'0'
NEXT2     DATA      F'0'
```

When the first RDCURSOR is issued, if the cursor is on the third line of the logical screen and ten spaces from the left margin, then, following the execution of the RDCURSOR, variable LN will contain 3 and variable COL will contain 10.

When the second RDCURSOR is executed, if the cursor is outside the logical screen, then both NEXT1 and NEXT2 will be set to a value of -1.

READ - Read records from a data set

The READ instruction retrieves one or more records from a disk, diskette, or tape data set and places them in a buffer area you define. You must allocate enough buffer space for the operation.

You can read disk or diskette data sets either sequentially or directly. These data sets are read in 256-byte record increments. The *Operator Commands and Utilities Reference* describes the format of a record created with the text editor, $FSEDIT. (For information on using 1024-byte-per-sector diskettes, see the *Installation and System Generation Guide*.) You can only read tape data sets sequentially. A READ operation for tape can retrieve a record from 18 to 32767 bytes long.

You have the option to place a disk read request at the top of the disk I/O chain. Such requests are made with the disk immediate read option. A disk immediate read request will be serviced before others in the chain. A coding example follows in this section. (Refer to "Coding Example - Disk Immediate Read" on page LR-381)

The READ instruction can take advantage of the cross-partition capability that enables your program to share data with a program or task in another partition. Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 contains an example of a cross-partition READ operation. You can find more information on cross-partition services in the *Event Driven Executive Language Programming Guide*.

Syntax:

```
label         READ      DSx,loc,count,relrecno|blksize,END=,
                        ERROR=,WAIT=,PREC=,P1=,P2=,P3=,P4=

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

Operand	***Description***
DSx	The data set from which you are reading. Code DSx, where "x" is a positive integer that indicates the relative position (number) of the data set in the list of data sets you defined on the PROGRAM statement. The value can range from 1 to the maximum number of data sets defined in the list. The maximum range is from 1-9. You can substitute a DSCB name defined by a DSCB statement for DSx.
loc	The label of the buffer area where the data is to be placed. When reading disk or diskette data sets, you must make sure that this area is a multiple of 256 bytes.

READ - Read records from a data set *(continued)*

When reading tape data sets, this area must equal or exceed the value you code for the blksize operand.

READ normally assumes the buffer is in the same partition as the currently executing program. You can read records into a buffer in another partition, however, by using the cross-partition capability of the READ instruction.

count The number of contiguous records to be read. If the program sets the field to 0, no I/O operation is performed. A count of the actual number of records read is returned in the second word of the task control block if WAIT=YES is coded. Note, however, if the incorrect blocksize is specified, the correct blocksize is 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 reads the remaining records and returns an end-of-data return code to the program.

relrecno The number of the record that is to be read from a disk or diskette data set. The record number is relative to the first record in the data set, and the numbering starts with 1. You can code a positive integer or the label of a data area containing the value.

You can request a sequential read operation by coding a 0 or by allowing this operand to default. If an end-of-data (EOD) indicator was previously set, an EOD is returned when the logical EOD is encountered. If the EOD indicator has not been set, the EOD returned represents the physical end-of-data.

A value other than 0 indicates that a direct READ is requested. An EOD indication is returned if an attempt is made to access a record outside the physical data set.

If you code a self-defining term, or an equated value indicated by a plus sign (+), then it is assumed to be a single-word value and is, therefore, generated as an inline operand. Because this is a one-word value, it is limited to a range of 1 to 32767 (X'7FFF').

If you code an indexable value or an address for this operand, the PREC operand can be used to further define whether relrecno is to be a single-word or double-word value.

PREC=D extends the maximum range of relrecno beyond the 32767 value to the limit of a double-word value (2147483647 or X'7FFFFFFF').

A sequential READ starts with relative record number 1 or the record number specified by a POINT instruction. The supervisor keeps track of sequential READ instructions and increments an internal next-record-pointer for each record read in sequential mode (relrecno is 0). Direct READ operations (relrecno is not 0) can be intermixed with sequential operations, but this does not change the next-record-pointer used by sequential operations.

READ - Read records from a data set *(continued)*

blksize The number of bytes to be read from a tape data set. The range is from 18 to 32767. You can code a self-defining term or the label of a data area containing the value. The default for this operand is 256 bytes of data. If you code 0 *or* do not code this operand, the instruction reads the default number of bytes.

The first word of the TCB contains 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 stored only in the second word of the TCB if you code WAIT=YES or allow the WAIT= operand to default to YES. If you code WAIT=NO and the blksize specification is incorrect, you can check the $DSCBR3 field in the DSCB for the actual number of records read or the actual blksize.

Do not code this operand in a READ instruction containing the relrecno operand.

PREC= This operand further defines the relrecno operand when you code an address or an indexable value for that operand. PREC=S (the default) limits the value of relrecno to single-word precision or to a value of 32767 (X'7FFF').

Coding PREC=D gives the relrecno operand a doubleword precision and extends the range of its maximum value to a doubleword value of 2147483647 (X'7FFFFFFF').

Do not code this operand in a READ instruction containing the blksize operand.

END= The label of the first instruction of the routine to be invoked if an end-of-data set condition is detected during the READ operation (return code=10). If you do not code this operand, the system treats an end-of-data set condition as an error.

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

Do not code this operand if you code WAIT=NO.

You can set or change the end-of-data by using the SE command of $DISKUT1. See *Operator Commands and Utilities Reference* for additional information.

ERROR= The label of the first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If you do not specify this operand, control passes to the instruction following the READ instruction and you must test the return code in the first word of the task control block for errors.

READ - Read records from a data set *(continued)*

Do not code this operand if you code WAIT=NO.

WAIT= YES (the default), to suspend the current task until the operation is complete.

NO, to return control to the current task after the operation is initiated. Your program must issue a subsequent WAIT DSx to determine when the operation is complete.

You cannot code the END and ERROR operands if you code WAIT=NO. You must subsequently test the return code in the Event Control Block (ECB) named DSx or in the first word of the task control block (TCB). The label of the TCB is the label of your program or task.

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 "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Examples

1) The following READ instruction reads a single 327-byte record from a standard label (SL) tape. If an end-of-data set tape mark is detected, control passes to the statement named END1. If an error occurs, control passes to the statement named ERR.

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

2) The following READ instruction does the same as in the previous example except that two records are read into your storage buffer (BUFF2). BUFF2 must be at least 654 bytes long.

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

READ - Read records from a data set *(continued)*

Coding Example - Read

The READ instruction in this example reads the next sequential record from the first relative data set specified in the list of data sets in the PROGRAM statement. If end-of-file is encountered during the read, the program passes control to the NOTFOUND label. If an unrecoverable I/O error is encountered, the program passes control to the label DSKRDERR. Otherwise, the instruction reads the record and places the data in the 256-byte buffer area labeled DISKBUFF.

```
READ     PROGRAM   LOOKUP,DS=(CHART1,CHART2)
LOOKUP   EQU       *
         READ      DS1,DISKBUFF,1,0,ERROR=DSKRDERR,END=NOTFOUND
         MOVEA     #1,DISKBUFF
         DO        16,TIMES
            IF        ((0,#1),EQ,$NAME,(16,BYTE)),GOTO,GOTNAME
            ENDIF
            ADD       #1,16
         ENDDO
         GOTO      LOOKUP
*
NOTFOUND EQU     *
         PRINTEXT '@EMPLOYEE FILE DOES NOT CONTAIN EMPLOYEE NAME '
         PRINTEXT $NAME
         GOTO    ENDIT
*
DSKRDERR EQU     *
         PRINTEXT '@UNRECOVERABLE DISK READ ERROR ON EMPLOYEE FILE'
         GOTO    ENDIT
*
GOTNAME  EQU     *
ENDIT    PROGSTOP
DISKBUFF BUFFER  265,BYTES
         ENDPROG
         END
```

READ - Read records from a data set *(continued)*

Coding Example - Disk Immediate Read

There are situations in which you have 1 or more applications already active on a Series/1 and desire to perform a disk read without having to wait for the completion of active programs. Use the disk immediate read option to make such requests. This special READ request is placed at the top of a disk I/O chain and serviced before other requests.

The following coding example illustrates how to code $DSCBPRI to set the priority read bit in the DSCB. Any READ request made directly after $DSCBPRI is set executes immediately. The bit resets automatically to continue operations normally as soon as that instruction prioritized for immediate execution is effected.

```
PROG1    PROGRAM   START
         COPY      DSCBEQU
         .
         .
         .
START    EQU
         .
         .
         .
         IOR       INDATA+$DSCBFLG,+$DSCBPRI   SET PRIORITY READ BIT IN DSCB
         READ      INDATA,BUF,1,1              READ A RECORD
         .
         .
         .
         ENDPROG
BUF      DC        128F'0'
         DSCB      DS#=INDATA,DSNAME=TEST
         END
```

READ - Read records from a data set *(continued)*

Disk and Tape Return Codes

Disk and 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.
- The first word of the task control block (TCB). The label of the TCB is the label of your program or task (taskname).

The possible return codes and their meaning for disk and tape are shown in tables later in this section.

If a tape error occurs, the read/write head positions itself immediately following the record in which the error occurred. This indicates that a retry has been attempted, but was unsuccessful. The count field, in the READ instruction, may or may not have been set to zero under this condition.

You can get detailed information on an error by using the $LOG utility to capture the I/O error. Refer to the *Problem Determination Guide* for information on how to use $LOG.

Note: If an error is encountered during a sequential I/O operation, the relative record number for the next sequential request is not updated. This can cause errors on all following sequential I/O operations.

READ - Read records from a data set *(continued)*

Disk/Diskette Return Codes

Return Code	Condition
-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	Read device status I/O instruction error.
5	Unrecoverable I/O error.
6	Error on issuing I/O instruction.
7	A no record found condition occurred, a seek for an alternate sector was performed, and another no record found occurred, for example, no alternate is assigned.
8	A system error occurred while processing an I/O request for a 1024-byte sector diskette.
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	Data set not open or device marked unusable when I/O was requested.
12	DSCB was not OPEN; DDB address = 0.
13	If extended deleted record support was requested ($DCSBFLG bit 3 on), the referenced sector was not formatted at 128 bytes/sector or the request was for more than one 256-byte sector. If extended deleted record support was not requested ($DSCBFLG bit 3 off), a deleted sector was encountered during I/O.
14	The first sector of the requested record was deleted.
15	The second sector of the requested record was deleted.
16	The first and second sectors of the requested record were deleted.
17	Cache fetch error. Contact your IBM customer engineer.
18	Bad cache error. Contact your IBM customer engineer.
24	End of tape.
30	Device not a tape.

READ - Read records from a data set *(continued)*

Tape Return Codes and Tape Post Codes

Return Code	Condition
-1	Successful completion.
1	Exception but no status.
2	Error reading cycle steal status.
3	I/O error; retry count exhausted.
4	Error issuing READ CYCLE STEAL STATUS.
6	I/O error issuing I/O operations.
10	End of data; a tape mark was read.
21	Wrong length record.
22	Device not ready.
23	File protected.
24	End of tape.
25	Load point.
26	Unrecoverable I/O error.
27	SL data set not expired.
28	Invalid blocksize.
29	Offline, in-use, or not open.
30	Incorrect device type.
31	Close incorrect address.
32	Block count error during close.
33	Close detected on EOV1.

The following post codes are returned to the event control block (ECB) of the calling program.

Post Code	Condition
-1	Function successful.
101	TAPEID not found.
102	Device not offline.
103	Unexpired data set on tape.
104	Cannot initialize BLP tapes.

READTEXT - Read text entered at a terminal

The READTEXT instruction reads an alphameric character string entered by the terminal operator.

The instruction can also print or display a prompt message to request input.

The supervisor places a return code in the first word of the task control block (taskname) whenever a READTEXT instruction causes a terminal I/O operation to occur. If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described under "Terminal I/O Return Codes" on page LR-394 and also in the *Messages and Codes*.

Syntax:

```
label        READTEXT loc,pmsg,PROMPT=,ECHO=,TYPE=,MODE=,
                      XLATE=,SKIP=,LINE=,SPACES=,CAPS=,
                      COMP=,PARMS=(parm1,...,parm8),MSGID=,P1=,P2=

Required:    loc
Defaults:    PROMPT=UNCOND,ECHO=YES,TYPE=DATA,MODE=WORD,
             XLATE=YES,SKIP=0,LINE=current line,SPACES=0
             MSGID=NO
Indexable:   loc,pmsg,SKIP,LINE,SPACES
```

Operand	*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 can be defined by DATA or DC statements, but you must adhere to the format of the TEXT statement. To satisfy the length specification, the input is either truncated or padded to the right with blanks, as necessary. The TEXT statement count field is also updated. If a static screen is in use, null characters are not translated into blanks. If the length specification is greater than the system buffer size, then the length is limited to the buffer size. If a user buffer is specified on an IOCB instruction and you have issued an ENQT to the corresponding terminal, then the user buffer size will apply to the input length. The loc operand may also be the label of a BUFFER statement referred to by an IOCB instruction. If this is the case, the input is direct; that is, 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 during system generation. Refer to the TERMINAL statement in the *Installation and System Generation Guide* for information on the default sizes.

READTEXT - Read text entered at a terminal *(continued)*

pmsg The prompt message. Code the label of a TEXT statement or an explicit text message enclosed in single quotes. The READTEXT instruction issues this prompt according to the parameter you code for the PROMPT keyword.

To retrieve a prompt message from a data set or module containing formatted program messages, code the number of the message you want displayed or printed. You must code a positive integer or a label preceded by a plus sign (+) that is equated to a positive integer. If you retrieve a prompt message from storage, you must also code the COMP= operand. See Appendix E, "Creating, Storing, and Retrieving Program Messages" on page LR-615 for more information.

PROMPT= COND (conditional), to prevent the system from displaying the prompt message if you enter text before the prompt.

UNCOND (unconditional), to have the system display the prompt message without exception. UNCOND is the default.

If you code PROMPT=COND without specifying a prompt message, the instruction does not wait for 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= 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 ECHO operand in READTEXT is equivalent to PROTECT=YES in other terminal I/O instructions.

YES (the default), to allow the input text to be printed on the terminal.

MODE= WORD (the default), to end the READTEXT operation when the system encounters a blank character (space). Leading blanks, however, are ignored. Lowercase input characters, including terminal control characters, are automatically converted to uppercase. The 3101 in block mode with a static screen separates all fields by blanks.

LINE, if the string to be read can include imbedded blanks. Any lowercase characters entered are left in lowercase.

For a 3101 in block mode with a static screen and with TYPE=ALL coded, a blank will precede each field.

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.

READTEXT - Read text entered at a terminal *(continued)*

For a 4978, 4979, or 4980 with a static screen, the READTEXT operation normally ends when the instruction fills the entire text field, when it reaches a protected field, or when it reaches the end of the logical line.

For 3101 in block mode, the READTEXT operation normally ends when the instruction fills the entire text field, or when it reaches the end of the screen. However, the TYPE operand determines what fields are read in.

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= The type of data to be transferred from a 4978, 4979, 4980, or a 3101 in block mode with a static screen.

When a READTEXT has been issued to a 3101 in block mode, any changed fields are reset to a unmodified condition.

Code TYPE=DATA (the default) to transfer only data fields.

Code TYPE=ALL to transfer both protected and data (unprotected) fields.

Code TYPE=MODDATA to transfer only those data fields which have been changed by the terminal operator (4978, 4980, or 3101 in block mode) for static screens.

Code TYPE=MODALL to transfer, along with each changed data field, the protected fields which precede it.

If coded for a 3101 in block mode with a static screen, TYPE=MODALL defaults to TYPE=MODDATA.

XLATE= NO, if the input line is not to be translated to EBCDIC. The character-delete and line-delete codes lose their special functions under this option, and MODE=LINE is implied. (See the *Communications Guide* for an explanation of 3101 Internal Code Representations.)

For a 3101 in block mode, terminal I/O support does not remove the escape sequences or attribute bytes from the data stream. Also, the TERMCTRL SET,ATTR or TERMCTRL SET,STREAM operands are ignored while the instruction executes.

Note: For a description of 3101 escape sequences, see *IBM 3101 Display Terminal Description*, GA18-2033.

If the terminal transmits characters in mirror image format and XLATE=NO is coded, the characters will be placed in storage in the terminal's native format.

READTEXT - Read text entered at a terminal *(continued)*

YES (the default), causes the supervisor to translate the terminal's binary code to EBCDIC, the standard Series/1 representation of data. Code XLATE=YES when you are coding a READTEXT instruction for Series/1-to-Series/1 communication.

SKIP= The number of lines to be skipped before the system does an I/O operation. For example, if your cursor is at line 2 on a display screen and you code SKIP=6, the system does the I/O operation on line 8. For a printer, the SKIP operand controls the movement of forms.

The SKIP operand causes the system to display or print the contents of the system buffer.

If you specify a value greater than or equal to the logical page size, the system divides this value by the page size and uses the remainder in place of the value you specify. For roll screens, the logical page size equals the screen's bottom margin minus the number of history lines and the screen's top margin.

LINE= The line number on which the system is to do an I/O operation. Code a value between zero and the number of the last usable line on the page or logical screen. The line count begins at the top margin you defined for the printer or display screen. LINE=0 positions the cursor at the top line of the page or screen you defined; LINE=1 positions the cursor at the second line of the page or screen. For roll screens, line 0 equals the screen's top margin plus the number of history lines.

For printers and roll screens, if you code a value less than or equal to the current line number, the system does the I/O operation at the specified line on the next page or logical screen. For static screens, if you code a value within the limits of the logical screen, the system does the I/O operation on the line you specified.

If you code a value greater than the last usable line number, the system divides this value by the logical page size and uses the remainder as the line number on which to do the I/O operation. For example, if you code LINE=22 and your roll screen has a logical page size of 20, the I/O operation occurs on the second line of the logical screen.

The LINE operand causes the system to print or display the contents of the system buffer.

SPACES= The number of spaces to indent before the system does an I/O operation. SPACES=0, the default, positions the cursor at the beginning of the left side of the page or screen. If the value you specify is beyond the limits of the logical screen or page, the system indents the next line by the excess number of spaces.

When you code the LINE or SKIP operands with SPACES, the system begins indenting from the left margin of the page or screen. If you specify SPACES

READTEXT - Read text entered at a terminal *(continued)*

without coding LINE or SKIP, the system begins indenting from the last cursor position on the line.

For an IBM 3101 in block mode, if no prompt message is specified, a READTEXT instruction will read data from the beginning of the screen and will ignore any cursor positioning by this operand.

CAPS= Converts EBCDIC data received in a READTEXT operation to uppercase characters. This operand is valid only for data that is defined by a TEXT or BUFFER statement.

Code CAPS=Y to convert all the data defined by a TEXT or BUFFER statement to uppercase characters. When specifying CAPS=Y, you must link-edit your program using the autocall feature of $EDXLINK.

To convert a specified number of bytes to uppercase, code that number with the CAPS operand. Capitalization starts from the first byte of the data received. For example, CAPS=3 capitalizes the first three bytes of data defined by the TEXT or BUFFER statement.

The count you specify should not exceed the length of the TEXT or BUFFER statement that contains the data. If the length is exceeded, the operation is still performed, but data beyond the TEXT or BUFFER statement may be modified.

When you code a value with the CAPS operand, the system does an inclusive OR (IOR) of a X'40' byte to each EBCDIC byte. (See Coding Example 2 at the end of this section.) A lowercase "a" (X'81'), for example, is converted to an uppercase "A" (X'C1'). Characters already capitalized remain unchanged. The IOR operation is done after the READTEXT instruction reads in the data.

Notes:

1. Coding XLATE=NO and the CAPS operand causes an assembly error.
2. If you specify MODE=WORD with the CAPS operand, the CAPS operand has no effect. MODE=WORD automatically converts lowercase input characters to uppercase.

COMP= The label of a COMP statement. You must specify this operand if the READTEXT instruction is retrieving a prompt message from a data set or module containing formatted program messages. The COMP statement provides the location of the message. (See the COMP statement description for more information.)

PARMS= The labels of data areas containing information to be included in a message you are retrieving from a data set or module containing formatted program messages. You can code up to eight labels. If you code more than one label, you must enclose the list in parentheses.

READTEXT - Read text entered at a terminal *(continued)*

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

MSGID= YES, if you want the message number and four-character prefix to be printed at the beginning of the message you are retrieving from a data set or module containing formatted program messages. See the COMP statement operand 'idxx' for a description of the four-character prefix.

NO (the default), to prevent the system from printing or displaying this information at the beginning of the message.

Note: To use this operand, you must have included the FULLMSG module in your system during system generation. Refer to *Installation and System Generation Guide* for a description of this module.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Advance Input Considerations

Input that you enter before a program requests it (advance input) normally resides in the system buffer. When your program issues a READTEXT instruction, the instruction immediately reads the advance input from the buffer. If a terminal output operation takes place just before the READTEXT operation, however, the READTEXT instruction does not read in the advance input. The instruction does not read the correct advance input because the terminal output operation has used the system buffer.

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

3101 Display Considerations

When using a 3101 in block mode, special considerations are required. The most recent TERMCTRL SET,ATTR or its default value determines both the attribute byte to be used for a prompting message and the field to be read. The TERMCTRL SET,ATTR, or its default value allows the fields for the prompt, if used, and the response to be programmed according to one of the attributes allowed by the 3101. After the data is read from the device, terminal I/O support will remove all the escape sequences from the 3101 data stream before transferring it to your application program. (For a description of 3101 escape sequences refer to the *IBM 3101 Display Terminal Description*, GA18-2033.)

In static screen mode, if there is no prompt message, the read will start from the beginning of the screen, regardless of any SKIP, LINE, or SPACES parameters in effect, because the 3101 in block mode does not have a direct read capability.

If a TERMCTRL SET,STREAM=YES is in effect, the data read is converted to EBCDIC. However, the escape sequences and attribute bytes are not removed from the data stream.

READTEXT - Read text entered at a terminal *(continued)*

A TERMCTRL SET,ATTR=NO, has no effect on input data.

Syntax Examples

1) Read text into the data area labeled OPTION. The prompt, 'ENTER OPTION' is conditional.

```
         READTEXT  OPTION,'ENTER OPTION: ',PROMPT=COND
            .
            .
OPTION   TEXT      LENGTH=2
```

2) Read text into the data area labeled NAME. The prompt, 'ENTER YOUR NAME', is unconditional.

```
         READTEXT  NAME,'ENTER YOUR NAME: '
            .
            .
NAME     TEXT      LENGTH=44
```

3) Read text into the data area labeled PASSWORD. The prompt, 'ENTER PASSWORD', is unconditional.

```
         READTEXT  PASSWORD,'ENTER PASSWORD: ',PROTECT=YES
            .
            .
PASSWORD TEXT      LENGTH=8
```

4) Read text into the data area labeled NEXTLINE. The text string can include imbedded blanks because MODE=LINE.

```
         READTEXT  NEXTLINE,MODE=LINE
            .
            .
NEXTLINE TEXT      LENGTH=80
```

Coding Examples

1) The following example uses a series of READTEXT instructions to set up a logon sequence for employees using an online time-sharing system.

The WELCOME message is displayed on the third line of the screen. This message is followed on the fifth line of the screen by a prompt message requesting entry of a LOGON command. The LOGMSG2 prompt always appears because PROMPT defaults to unconditional. An unconditional PROMPT is then displayed requesting entry of an employee number. If a blank is entered the logon process ends. Otherwise, the code verifies that the employee number is six digits long. If the employee number is not six digits, a branch to EMPLOYEE causes a retry.

READTEXT - Read text entered at a terminal *(continued)*

The READTEXT for the password is conditional so that the prompt is not displayed if there is advanced input accompanying a proper length I.D. number. The READTEXT contains the MODE=LINE keyword so that the text can contain embedded blanks.

A proper match of I.D. and password is made by calling subroutine CHKPASS. A correct match causes a branch to the GOODPASS label; otherwise, the next sequential instruction is executed which is the beginning of an error routine. A maximum of four incorrect passwords are examined for each logon attempt. If logon is not successful by the fourth attempt, the process ends.

If the logon is accepted, a READTEXT is issued for a title line. This title line is used on system reports which are produced during the current logon session.

```
LOGON    EQU      *
         PRINTEXT LOGMSG1,LINE=3,SPACES=35
         READTEXT LOGCMD,LOGMSG2,LINE=5,SPACES=35
*
EMPLOYEE EQU      *
         READTEXT EMPNUM,'@ENTER YOUR EMPLOYEE NUMBER'
         IF       (EMPNUM,EQ,BLANK,(1,BYTE)),GOTO,LOGON
         IF       (EMPNUM-1,NE,6),GOTO,EMPLOYEE
*
GETPASS  EQU      *
         READTEXT PASSWORD,'@ENTER PASSWORD',PROMPT=COND,MODE=LINE,   X
               TYPE=ALL
*
*                       VERIFY I.D. NUMBER & PASSWORD
*
         CALL     CHKPASS
         IF       (PASSCHK,EQ,-1),GOTO,GOODPASS
*
BADPASS  EQU      *
         PRINTEXT 'INVALID PASSWORD FOR USERID',SKIP=1
         PRINTEXT EMPNUM
         ADD      BADWORD,1
         IF       (BADWORD,LT,4),GOTO,GETPASS
         MOVE     BADWORD,0
         GOTO     LOGON
            .
            .
         SUBROUT  CHKPASS
            .
         MOVE     PASSCHK,-1
         RETURN
            .
GOODPASS EQU      *
         READTEXT TITLE,TITLEMSG,SKIP=1,MODE=LINE
            .
            .
            .
LOGMSG1  TEXT     ' WELCOME TO ONLINE TIME SHARING'
LOGMSG2  TEXT     ' PLEASE ENTER YOUR LOGON COMMAND'
LOGCMD   TEXT     LENGTH=2
EMPNUM   TEXT     LENGTH=6
PASSWORD TEXT     LENGTH=3
```

READTEXT - Read text entered at a terminal *(continued)*

```
TITLE    TEXT     LENGTH=60
TITLEMSG TEXT     'ENTER A 60 CHARACTER TITLE FOR                    X
               YOUR REPORTS'
BADWORD  DATA     F'0'
BLANK    DATA     C' '
PASSCHK  DATA     F'0'        CODE WORD TO INDICATE
*                             VALIDITY OF PASSWORD
```

2) When you code a value with the CAPS operand, the system generates an IOR instruction to capitalize the specified data. The following example shows the use of the CAPS operand and how you can achieve the same results by coding an IOR instruction directly in your application program.

With the CAPS operand

```
               .
         READTEXT  A,CAPS=5,MODE=LINE
               .
               .
A        TEXT      LENGTH=5
```

Without the CAPS operand

```
               .
         READTEXT  A
         IOR       A,X'40',(5,BYTES)
               .
               .
A        TEXT      LENGTH=5
```

3) In this example, the READTEXT instruction displays a prompt message contained in MSGMOD, a storage-resident message area. Because +MSG8 equals 8, the system retrieves the eighth message in MSGMOD.

```
         READTEXT  NAME,+MSG8,PROMPT=COND,COMP=MSGSTMT
               .
MSG8     EQU      8
               .
         PROGSTOP
NAME     TEXT     LENGTH=8
MSGSTMT  COMP     'SRCE',MSGMOD,TYPE=STG
```

READTEXT - Read text entered at a terminal *(continued)*

Message Return Codes

The system issues the following return codes when you retrieve a prompt message from a data set or module containing formatted program messages. The return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Message successfully retrieved
301-316	Error while reading message from disk. Subtract 300 from this value to get the actual return code. See the disk return codes following the READ or WRITE instruction for a description of the code.
326	Message number out of range
327	Message parameter not found
328	Instruction does not supply message parameter(s)
329	Invalid parameter position
330	Invalid type of parameter
331	Invalid disk message data set
332	Disk message read error
333	Storage-resident module not found
334	Message parameter output error
335	Disk messages not supported (MINMSG support only)

Terminal I/O Return Codes

The terminal I/O return codes are all listed here and following the PRINTEXT instruction. A complete list of all return codes also can be found in the *Messages and Codes*. You must select the group of codes that represents the particular device type you are using. A list of the terminal I/O return code groups follows:

- General Terminal I/O
- Virtual Terminal
- ACCA Devices
- Interprocessor Communication
- General Purpose Interface Bus
- Series/1-to-Series/1 Adapter

READTEXT - Read text entered at a terminal *(continued)*

General Terminal I/O Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-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.
8	Printer power has been switched off and switched back on or a power failure has occurred.
>10	A code greater than 10 can indicate 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 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

If the return code is for devices supported by IOS2741 (2741, PROC) and a code greater than 128 is returned, subtract 128; the result then contains status word 1 of the ACCA. Refer to the *IBM Series/1 Communications Features Description*, GA34-0028 for determination of the special error condition.

READTEXT - Read text entered at a terminal *(continued)*

Virtual Terminal Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Transmit Condition	Receive Condition
X'8Fnn'	Not applicable.	LINE=nn received.
X'8Enn'	Not applicable.	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.

A further description of the virtual terminal return codes follows:

LINEnn (X'8Fnn'): Returned for a READTEXT or GETVALUE instruction if the other program issued an instruction with a LINE= operand. This operand tells the system to do an I/O operation on a certain line of the page or screen. The return code enables the receiving program to reproduce on an actual terminal the output format intended by the sending program.

SKIPnn (X'8Enn'): The other program issued an instruction with a SKIP= operand. This operand tells the system to skip several lines before doing an I/O operation.

Line Received (-2): Indicates that an instruction (usually READTEXT or GETVALUE) has sent information but has not issued a carriage return to move the cursor to the next line. The information is usually a prompt message.

New Line Received (-1): Indicates transmission of a carriage return at the end of the data. The cursor is moved to a new line. This return code and the Line Received return code help programs to preserve the original format of the data they are transmitting.

Not attached (1): A virtual terminal does not or cannot refer to another virtual terminal.

Disconnect (5): The other virtual terminal program ended because of a PROGSTOP or an operator command.

Break (8): Indicates that both virtual terminal programs are attempting to do the same type of operation. When one program is reading (READTEXT or GETVALUE), the return code means the other program has stopped sending and is waiting for input. When one program is writing (PRINTEXT or PRINTNUM), the return code means the other program is also attempting to write.

If you defined only one virtual terminal with SYNC=YES, then that task always receives the break code. If you defined both virtual terminals with SYNC=YES, then the task that last attempted the operation receives the break code.

READTEXT - Read text entered at a terminal *(continued)*

ACCA Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful completion.
1-08	Return code for last operation placed in information status byte (ISB). Refer to the hardware description manual for status on the device you are using.
11	Write operation (I/O complete).
12	Read operation (I/O complete).
14,15	Condition code +1 after I/O start or condition code after I/O complete.

Interprocessor Communication Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

CODTYPE=	Return Code	Condition
EBCDIC	FDFF	End of transmission (EOT).
EBCDIC	FEFF	End of record (NL).
EBCDIC	FCFF	End of subrecord (EOSR).
EBCD/CRSP	1F	End of transmission (EOT).
EBCD/CRSP	5B	End of record (NL).
EBCD/CRSP	(none)	End of subrecord (EOSR).

General Purpose Interface Bus Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful completion.
1	Device not attached.
2	busy condition.
3	busy after reset.
4	command reject.
6	Interface data check.
256 + ISB	Read exception.
512 + ISB	Write exception.
1024	Attention received during an operation (may be combined with an exception condition).

READTEXT - Read text entered at a terminal *(continued)*

Series/1-To-Series/1 Return Codes

The return codes are returned in the first word of the task control block of the program issuing the instruction.

Return Code	Condition
-1	Successful.
1	Device not attached.
2	System error (busy condition).
3	System error (busy after reset).
4	System (command reject).
5	Device not ready (not reported for S/1 - S/1).
6	Interface data check.
7	Overrun recieved (not reported for S/1 - S/1).
138, 154	An error has occurred that can only be determined by displaying the device cycle steal status word with the TERMCTRL STATUS function and checking the bits to determine the cause of the error.
1002	Other system not active.
1004	Checksum error detected.
1006	Invalid operation code or sequence.
1008	Timeout on data transfer.
1010	TERMCTRL ABORT issued by responding processor.
1012	Device reset (TERMCTRL RESET) issued by the other processor.
1014	Microcode load to attachement failed during IPL.
1016	Invalid or unsolicited interrupt occurred.
1050	TERMCTRL ABORT issued and no operation pending.
1052	TERMCTRL IPL attempted by slave processor.
1054	Invalid data length.

RESET - Reset an event or process interrupt

The RESET instruction resets 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 subsequently issues 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 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 before 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 done physically by setting the first word of its ECB to 0.

Note: Specify the address key of an event to be reset with the task target address key, $TCBADS.

Syntax:

```
label        RESET    event,P1=

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

Operand	*Description*
event	The label 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 operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

RESET - Reset an event or process interrupt *(continued)*

Coding Example

The RESET instruction at label RES1 refers to a specific ECB and can operate only on the ECB labeled ECB1.

The RESET instruction at label RES2 uses the parameter naming operand, P1=, to supply the address of the ECB on which RESET is to operate. The application program then ensures that the address of the ECB that is to be cleared is moved to the label named by the P1= operand in the RESET instruction.

```
                 .
                 .
                 .
RES1      RESET   ECB1
          WAIT    ECB1
                 .
                 .
          MOVEA   ANYECB,WAITECB
RES2      RESET   ECB1,P1=ANYECB
                 .
                 .
ECB1      ECB
WAITECB   ECB
                 .
                 .
                 .
```

RETURN - Return to the calling program

The RETURN instruction provides linkage back to a calling program from a subroutine. Each subroutine must contain at least one RETURN instruction.

Syntax:

```
label          RETURN

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

Operand	*Description*
none	none

Coding Example

In the example, each of the three RETURN instructions at labels RET1, RET2, and RET3 causes task execution to resume at the instruction following the RETURN1 label. This occurs because each of the instructions passes control to the instruction following the subroutine call.

```
         .
         .
         .
         CALL     DISKERR,MSGNUMBR
RETURN1  EQU      *
         .
         .
         SUBROUT DISKERR,MSGNO
         IF (MSGNO,EQ,1)
            PRINTEXT '@ DISK DATA SET HAS REACHED END-OF-FILE'
RET1        RETURN
         ELSE
            IF (MSGNO,EQ,2)
               PRINTEXT '@ DISK DATA SET IS NOT CATALOGUED'
RET2           RETURN
            ELSE
               PRINTEXT '@ DISK DATA SET IS READ-ONLY'
            ENDIF
         ENDIF
RET3     RETURN
```

SBIO - Specify a sensor-based I/O operation

The SBIO instruction specifies the sensor-based I/O operation you want to perform.

The instruction has a separate format for analog input, analog output, digital input, and digital output operations. Each of these formats is shown on the following pages.

Options available with the SBIO instruction allow you to:

- Automatically index using a previously defined BUFFER statement.
- Automatically update a buffer address after each operation.
- Use a short form of the instruction, omitting the "loc" operand (data location), to imply a data address within the SBIO control block.

You can also provide PULSE output and manipulate portions of the 16-bit I/O group with the BITS=(u,v) keyword.

The SBIO instruction refers to a three-to-four-character device label assigned with an IODEF statement. The IODEF statement contains the actual hardware address and the attributes you defined for the I/O device. (See IODEF for a description of how to code the statement.)

SBIO Control Block

Each IODEF statement you code creates a sensor-based input/output control block (SBIOCB) in your application program. The SBIOCB acts as a link between the SBIO operation and the device information contained in the IODEF statement. The SBIOCB, which contains a data I/O area and an event control block (ECB), also serves as a location where the supervisor can either store data (for AI and DI operations) or can fetch data (for AO and DO operations).

When your program executes an SBIO instruction, the supervisor either reads or writes data from or to a location in the IOCB with the label of a specified I/O point (for example, AI1, DI2, DO33, AO1). An application program can refer to these locations in the same way it refers to any other variable. This fact allows you to use the short form of the SBIO instruction (for example, SBIO DI1) and to refer to the label (DI1) in other instructions. You can equate device labels with more descriptive labels. For example, you could equate the device label DI15 with the label SWITCH as follows:

```
SWITCH EQU  DI15
```

You must code the device label, however, in the SBIO instruction.

Each control block also contains an ECB to be used by those operations that require the supervisor to respond to an interrupt and to "post" an operation as complete. Such operations include analog input (AI), process interrupt (PI), and digital I/O with external synchronization (DI/DO). For process interrupt, the label on the ECB is the same as the symbolic I/O point (PI3, for example). For analog and digital I/O, the label is the same as the symbolic I/O point with the suffix "END" (for example, DIxEND).

SBIO - Specify a sensor-based I/O operation *(continued)*

SBIO Analog Input

Syntax:

```
 label      SBIO       AIx,ERROR=,P1=
              or
 label      SBIO       AIx,loc,ERROR=,P1=,P2=
              or
 label      SBIO       AIx,loc,INDEX,EOB=,ERROR=,P1=,P2=
              or
 label      SBIO       AIx,loc,opnd3,SEQ=,ERROR=,P1=,P2=,P3=

Required:   AIx
Defaults:   no indexing, SEQ=NO
Indexable:  loc
```

Operand	***Description***
AIx	The label you assigned to an analog input device on the associated IODEF statement. AIx acts as the label of a single data storage location if you do not specify the loc operand.
loc	Buffer address or location where the system will store analog input. If you do not code the loc operand, the supervisor stores data from the operation in the SBIOCB created for the instruction.
EOB=	You can use this operand for buffer operations with automatic indexing. Code the label of a branch to be taken if: 1. The SBIO operation uses the last element of the buffer you defined. A return code of $OK is placed in the task name. 2. The buffer is full when the SBIO operation begins. The branch occurs without executing the SBIO instruction and the system places a return code of $BFRPFE in the task name. **Note:** If your program branches to the label you defined, you must reset the buffer count.
opnd3	Code INDEX to specify that the system is to do automatic indexing of a buffer you defined. You must define the buffer with a BUFFER statement. If you code a label or a constant for opnd3, the operand is the number of consecutive AI points to be used in the operation or the number of times to

SBIO - Specify a sensor-based I/O operation *(continued)*

repeat the operation on the same point. The SEQ operand determines the function of the operand.

SEQ= NO (the default), to repeat the operation on the same point the number of times indicated by opnd3.

YES, to use the number of consecutive AI points indicated by opnd3 in the operation.

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 you select. Bits 13 – 15 of this word contain the binary number representing the range of the AI reading. Bit 12 is zero. (Refer to the *IBM Series/1 4982 Sensor Input/Output Unit Description*, GA34-0027 for a detailed discussion of the analog-to-digital conversion.)

ERROR= The label of the instruction to be executed if the SBIO instruction is unsuccessful after two retries. If you do not code ERROR=, execution proceeds sequentially. In either case, the first word of the task control block contains the return code.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

This example shows a sensor-based I/O operation using the SBIO instruction and an IODEF statement to 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 POINTS INTO
*                                      NEXT 2 LOCATIONS OF 'BUF'
*
         SBIO   AI1,BUF,2,SEQ=NO       READ THE SAME POINT TWO TIMES
                                       AND PUT THE INFORMATION IN TWO
                                       LOCATIONS OF 'BUF'
```

Return Codes

The return codes for all SBIO instruction formats are listed under "SBIO (Digital Output)" on page LR-410.

SBIO - Specify a sensor-based I/O operation *(continued)*

SBIO (Analog Output)

Syntax:

```
label       SBIO       AOx,ERROR=,P1=
              or
label       SBIO       AOx,loc,ERROR=,P1=,P2=
              or
label       SBIO       AOx,loc,INDEX,EOB=,ERROR=,P1=,P2=

Required:   AOx
Defaults:   no indexing
Indexable:  loc
```

Operand	*Description*
AOx	The label you assigned to an analog output device on the associated IODEF statement. AOx acts as the label of a single data storage location if you do not specify the loc operand.
loc	An explicit constant or the address of the location of the output data. If you do not code the loc operand, the supervisor fetches data from the SBIOCB created for the instruction.
EOB=	You can use this operand for buffer operations with automatic indexing. Code the label of a branch to be taken if: 1. The SBIO operation uses the last element of the buffer you defined. A return code of $OK is placed in the task name. 2. The buffer is logically empty when the SBIO operation begins. The branch occurs without executing the SBIO instruction and the system places a code of $BFRPFE in the task name. **Note:** If your program branches to the label you defined, you must reset the buffer count.
opnd3	Code INDEX to specify that the system is to do automatic indexing of a buffer you defined. You must define the buffer with a BUFFER statement.
ERROR=	The label of the instruction to be executed if the SBIO instruction is unsuccessful after two retries. If you do not code ERROR=, execution proceeds sequentially. In either case, the first word of the task control block contains the return code.

SBIO - Specify a sensor-based I/O operation *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

This example shows a sensor-based I/O operation using the SBIO instruction and an IODEF statement to write analog output.

```
         IODEF AO1,ADDRESS=63
*
         SBIO  AO1                          SET AO1 TO VALUE IN 'AO1'
         SBIO  AO1,DATA                     SET AO1 TO VALUE IN 'DATA'
         SBIO  AO1,1000                     SET AO1 TO 1000
         SBIO  AO1,(0,#1)                   SET AO1 TO VALUE IN (0,#1)
         SBIO  AO1,BUF,INDEX                SET AO1 TO VALUE IN NEXT
```

Return Codes

The return codes for all SBIO instruction formats are listed under "SBIO (Digital Output)" on page LR-410.

SBIO - Specify a sensor-based I/O operation *(continued)*

SBIO (Digital Input)

Syntax:

```
label       SBIO       DIx,ERROR=,P1=
             or
label       SBIO       DIx,loc,ERROR=,P1=,P2=
             or
label       SBIO       DIx,loc,INDEX,EOB=,ERROR=,P1=,P2=
             or
label       SBIO       DIx,loc,BITS=(u,v),LSB=,ERROR=,P1=,P2=
             or
label       SBIO       DIx,loc,opnd3,ERROR=,P1=,P2=,P3=

Required:   DIx
Defaults:   no indexing,LSB=15
Indexable:  loc
```

Operand	***Description***
DIx	The label you assigned to a digital input device on the associated IODEF statement. DIx acts as the label of a single data storage location if you do not specify the loc operand.
loc	Buffer address or location where the system will store digital input. If you do not code the loc operand, the supervisor stores data from the operation in the SBIOCB created for the instruction.
EOB=	You can use this operand for buffer operations with automatic indexing. Code the label of a branch to be taken if: 1. The SBIO operation uses the last element of the buffer you defined. A return code of $OK is placed in the task name. 2. The buffer is full when the SBIO operation begins. The branch occurs without executing the SBIO instruction and the system places a code of $BFRPFE in the task name. **Note:** If your program branches to the label you defined, you must reset the buffer count.
opnd3	Code INDEX to specify that the system is to do automatic indexing of a buffer you defined. You must define the buffer with a BUFFER statement.

SBIO - Specify a sensor-based I/O operation *(continued)*

If opnd3 is the label of a variable or a constant representing the count of external synchronization read cycles, you must specify EXTSYNC (external synchronization) in the associated IODEF statement. Specifying EXTSYNC also provides a latched DI operation. The system reads the entire 16-bit group.

If you specify EXTSYNC on the IODEF statement but do not code opnd3, the system does a single unsynchronized I/O operation.

BITS=(u,v) The portion of a DI group 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= Input data is right justified to this bit with all unused bits set to 0. Code this operand only if you coded BITS=. The default is bit 15.

ERROR= The label of the instruction to be executed if the SBIO instruction is unsuccessful after two retries. If you do not code ERROR=, execution proceeds sequentially. In either case, the first word of the task control block contains the return code.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Example

This example shows a sensor-based I/O operation using the SBIO instruction and three IODEF statements to 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
```

SBIO - Specify a sensor-based I/O operation *(continued)*

Return Codes

The return codes for all SBIO instruction formats are listed under "SBIO (Digital Output)" on page LR-410.

SBIO - Specify a sensor-based I/O operation *(continued)*

SBIO (Digital Output)

Syntax:

```
label       SBIO      DOx,ERROR=,P1=
              or
label       SBIO      DOx,loc,ERROR=,P1=,P2=
              or
label       SBIO      DOx,loc,INDEX,EOB=,ERROR=,P1=,P2=
              or
label       SBIO      DOx,loc,BITS=(u,v),LSB=,ERROR=,P1=,P2=
              or
label       SBIO      DOx,loc,opnd3,ERROR=,P1=,P2=,P3=
              or
label       SBIO      DOx,(PULSE,dir),ERROR=

Required:   DOx
Defaults:   no indexing,LSB=15
Indexable:  loc
```

Operand	*Description*
DOx	The label you assigned to a digital output device on the associated IODEF statement. DOx acts as the label of a single data storage location if you do not specify the loc operand.
loc	An explicit constant or the address of the location of the output data. If you do not code the loc operand, the supervisor fetches data from the SBIOCB created for the instruction.
EOB=	You can use this operand for buffer operations with automatic indexing. Code the label of a branch to be taken if:

1. The SBIO operation uses the last element of the buffer you defined. A return code of $OK is placed in the task name.

2. The buffer is logically empty when the SBIO operation begins. The branch occurs without executing the SBIO instruction and the system places a code of $BFRPFE in the task name.

Note: If your program branches to the label you defined, you must reset the buffer count.

SBIO - Specify a sensor-based I/O operation *(continued)*

opnd3 Code INDEX to specify that the system is to do automatic indexing of a buffer you defined. You must define the buffer with a BUFFER statement.

If you specify a label or constant for opnd3, external synchronization is used.

BITS=(u,v) 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. This does not affect the condition of the other bits in the 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 IODEF statement.

LSB= Output data is taken from the output word with this bit being the least significant bit. Use this operand only if you coded BITS=. The default is bit 15.

(PULSE,dir) Code this operand to generate a pulse on the digital output group or subgroup you specified. Allowable directions (dir) are ON (or UP) and OFF (or DOWN).

ERROR= The label of the instruction to be executed if the SBIO instruction is unsuccessful after two retries. If you do not code ERROR=, execution proceeds sequentially. In either case, the first word of the task code block contains the return code.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Coding Examples

1) This example uses the SBIO instruction and three IODEF statements to 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
```

SBIO - Specify a sensor-based I/O operation *(continued)*

2) This example shows 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
```

Return Codes

You can find the return code for an SBIO operation by referring to the first word in the task control block (TCB). The label of the TCB is the label of your program or task (taskname).

Each condition shown below has a return code and an equate for that condition. If you refer to the equate in your program rather than the actual return code, your source code will always be current. You can obtain these equates when using $EDXASM by coding COPY ERRORDEF before the ENDPROG statement in your program.

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	AI invalid count field (AI/DI/DO count)
102	$BFRPFE	Buffer previously full or empty (indexing)
104	$DCMDREJ	Delayed command reject

In the following example, the program branches to label REDO if the condition "AI over voltage" occurs. The program refers to the equate $OVRVOLT. Note the use of the leading plus sign (+) with the equate to specify that it is a constant.

```
       SBIO AI1,ERROR=AIERR
         .
         .
         .
AIERR  IF    (taskname,EQ,+$OVRVOLT),GOTO,REDO
```

SCREEN - Convert graphic coordinates to a text string

The SCREEN instruction converts the x and y coordinates that represent a point on a screen to a four-character text string that becomes the graphic address of the point.

Syntax:

```
label       SCREEN    text,x,y,CONCAT=,ENHGR=,P1=,P2=,P3=

Required:   text,x,y
Defaults:   CONCAT=NO,ENHGR=NO
Indexable:  none
```

Operand	*Description*
text	Location of a text string at least four characters long.
x,y	Screen coordinates of a point to be translated. The range is 0 – 1023 for the full width of the screen and 0 – 779 for the screen height. You can extend this range by coding the ENHGR operand. Operands x and y can be locations containing data or explicit values, but both must be of the same type.
CONCAT=	Code CONCAT=YES to concatenate the results of the operation to the contents in text. The text string length is increased by four or by five if you code ENHGR=YES. The length of the text string is set to five if you code CONCAT=NO and ENHGR=YES. If you code CONCAT=NO and ENHGR=NO, the length of the text string is set to four.
ENHGR=	YES, to extend the range for the full width of the screen to 0 – 4095 and to extend the range for the screen height to 0 – 3120. When you code ENHGR=YES, a five-character graphic instruction is compiled. NO (the default), not to extend the range for the screen width or height.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Convert coordinates 520 and 300 to a text string. Concatenate the string to the contents of TEXT1.

```
SCREEN     TEXT1,520,300,CONCAT=YES
```

SETBIT - Set the value of a bit

The SETBIT instruction sets the value of a bit to 1 or 0. The bit is "on" if it contains a 1 and "off" if it contains a 0.

You can test to see if a bit is "on" or "off" with the IF instruction. The DO instruction allows your program to do a loop while or until a certain bit is "on" or "off".

Syntax:

```
label        SETBIT     data1,data2,ON|OFF,P1=,P2=

Required:    data1,data2,ON or OFF
Defaults:    none
Indexable:   data1,data2
```

Operand	*Description*
data1	The label of a data string that contains the bit to be set to 1 or 0.
data2	The location in data1 of the bit to be changed. You can code: • An integer or the label of an integer from 1 to 32767. • A hexadecimal value or the label of a hexadecimal value from 1 to 65535 (X'FFFF'). Bit 0 is the left-most bit of the data area.
ON	Sets the value of the bit to 1.
OFF	Sets the value of the bit to 0.

SETBIT - Set the value of a bit *(continued)*

Syntax Examples

1) Turn on the fifth bit in CONTROL.

```
          SETBIT    CONTROL,BIT,ON
            .
            .
            .
BIT       DATA      F'4'
```

2) Turn off the third bit in CONTROL.

```
          SETBIT    CONTROL,2,OFF
```

3) Turn on bit 15 in STATUS.

```
          SETBIT    STATUS,BIT,ON
            .
            .
            .
BIT       DATA      X'000E'
```

SHIFTL - Shift data to the left

The SHIFTL instruction shifts the contents of operand 1 to the left by the number of bit positions specified in 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.

Note: The precision of opnd2 should not exceed the precision of opnd1.

Syntax:

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

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

Operand	*Description*
opnd1	The label of a data area containing the data to be shifted left. You cannot code a self-defining term.
opnd2	The value by which the first operand is shifted. Code a self-defining term or the label of a data area.
count	The number of consecutive values in opnd1 on 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 can take one of the following forms: BYTE -- Byte precision WORD -- Word precision DWORD -- Doubleword precision
RESULT=	The label of a data area or vector in which the result is to be placed. If you code this operand, opnd1 is not modified.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

SHIFTL - Shift data to the left *(continued)*

Syntax Example

The SHIFTL instruction in this example changes the value in the data area labeled A from X'82F1' to X'0BC4' by shifting the bit string two positions to the left.

```
         .
         .
         SHIFTL A,2
         .
         PROGSTOP
A        DATA   X'82F1'     binary 1000 0010 1111 0001
         .
         .
```

After the operation, A equals:

Hexadecimal -- X'0BC4'

Binary -- 0000 1011 1100 0100

SHIFTR - Shift data to the right

The SHIFTR instruction shifts the contents of operand 1 to the right by the number of bit positions specified in 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.

Note: The precision of opnd2 should not exceed the precision of opnd1.

Syntax:

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

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

Operand	*Description*
opnd1	The label of the data area to be shifted. You cannot code a self-defining term.
opnd2	The value by which the first operand is shifted. Code a self-defining term or the label of a data area.
count	The number of consecutive values in opnd1 on 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 can take one of the following forms: BYTE -- Byte precision WORD -- Word precision DWORD -- Doubleword precision
RESULT=	The label of a data area or vector in which the result is to be placed. If you code this operand, opnd1 is not modified.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

SHIFTR - Shift data to the right *(continued)*

Syntax Example

The SHIFTR instruction in this example shifts the contents of C 24 bits to the right and stores the result of the operation in the data area labeled E. The value in C remains the same.

```
          .
          .
          SHIFTR C,24,DWORD,RESULT=E
          .
          PROGSTOP
C         DATA   X'A794B109'
E         DATA   X'00000000'
          .
          .
```

Before:

C = X'A794B109'
or
binary 1010 0111 1001 0100 1011 0001 0000 1001

E = X'00000000'
or
binary 0000 0000 0000 0000 0000 0000 0000 0000

After:

C = X'A794B109'
or
binary 1010 0111 1001 0100 1011 0001 0000 1001

E = X'000000A7'
or
binary 0000 0000 0000 0000 0000 0000 1010 0111

SPACE - Insert blank lines in a compiler listing

The SPACE statement inserts one or more blank lines in a compiler listing.

Because this statement does not generate code or constants in the object program, it can be placed between executable instructions in your source statement data set.

Syntax:

```
blank        SPACE    value

Required:    none
Defaults:    value = 1
```

Operand	*Description*
value	A positive integer specifying the number of blank lines to be inserted. If no value is entered, the system inserts one blank. If the value exceeds the number of lines remaining on the page, the statement has the same effect as an EJECT statement.

Coding Example

See the PRINT statement for an example using SPACE.

SPECPIRT - Return from Process Interrupt Routine

The SPECPIRT instruction returns control to the supervisor from a special process interrupt (SPECPI) routine that you provide. If the routine is in partition 1, control returns to the supervisor with a branch instruction. To return to the supervisor from another partition, your routine must execute a Series/1 assembler SELB instruction after registers R0 and R3 are saved in the level status block (LSB) you select.

You can use SPECPIRT only when you specify TYPE=BIT on the IODEF (Process Interrupt) statement.

```
label          SPECPIRT

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

Operand	*Description*
none	none

SQRT - Find the square root

The SQRT instruction finds 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. To use the SQRT instruction you must link-edit your program with $EDXLINK and specify $$SQRT,ASMLIB on an INCLUDE statement.

Syntax:

```
 label       SQRT     rsq,root,rem,P1=,P2=,P3=

Required:   rsq,root,rem
Defaults:   none
Indexable:  none
```

Operand	***Description***
rsq	The label 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 label of a 1-word data area where the square root is to be stored.
rem	The label of a 1-word data area where the remainder is to be stored.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Calculate the square root of the integer value in VALUE.

```
GETSQRT  EQU      *
         SQRT     VALUE,ROOT,REMAIN
                  .
                  .
                  .
VALUE    DATA     D'0'
ROOT     DATA     F'0'
REMAIN   DATA     F'0'
```

If the data area labeled VALUE contains the number 18611 (X'00004863'), the SQRT instruction would place a result of 136 (X'0088') in ROOT and a remainder of 115 (X'0073') in REMAIN.

STATUS - Set fields to check host status data set

The STATUS instruction defines the fields required to refer to a record in the "System Status Data Set" on the host computer.

TP SET, TP FETCH, and TP RELEASE refer to the label of the STATUS instruction. See the *Communications Guide* for information on how to use the System Status Data Set.

Syntax:

```
label          STATUS  index,key,length,P1=,P2=,P3=

Required:      label,index,key
Defaults:      length=0
Indexable:     none
```

Operand	***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. We suggest that a unique index be specified for each Series/1, but this is not a requirement.
key	A 1 - 8 alphameric 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 "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Coding Example

The following coding example shows a use of the STATUS instruction. The host communications facility (HCF) is required to execute the TP instructions that are used in this example.

STATUS - Set fields to check host status data set *(continued)*

In this example, a Series/1 program (PROGA) creates a message and sends it to the host computer. The sending Series/1 then waits for another Series/1 program (PROGB, possibly from a different Series/1) to receive the message and acknowledge the receipt by deleting the message.

The STATUS instruction in PROGA, at label STATUSA, defines the index and key needed to refer to a record. The TP SET instruction at label BEGINA makes an entry in the system status data. After creating the entry, PROGA goes into a loop of TP FETCH instructions that ends when the entry is not found.

The STATUS instruction in PROGB, at label STATUSB, defines the same index and key defined in PROGA. PROGB executes a TP FETCH instruction, at label TPB1, in an attempt to fetch the system status data set entry which it defined by the STATUS instruction parameters at label STATUSB.

If PROGA has not yet created the entry (through execution of the TP SET instruction at label BEGINA), an error occurs and PROGB will loop through the TP FETCH instruction until it does find an entry with the required index and key. After finding the entry, the TP RELEASE instruction deletes it and executes a PROGSTOP.

Deleting the entry causes the TP FETCH instruction in PROGA to take the error exit. PROGA then executes a PROGSTOP and ends.

```
PROGA     PROGRAM  BEGINA
STATUSA   STATUS   PROGID,KEYSTRNG
BEGINA    EQU      *
          TP       SET,STATUSA
TPLOOPA   EQU      *
          TP       FETCH,STATUSA,ERROR=ENDIT
          GOTO     TPLOOPA
ENDIT     PROGSTOP
             .
             .
             .
          ENDPROG
          END


PROGB    PROGRAM   TPLOOP
STATUSB  STATUS    PROGID,KEYSTRNG
TPLOOP   EQU       *
TPB1     TP        FETCH,STATUSB,ERROR=TPLOOPB
TPB2     TP        RELEASE,STATUSB
ENDALL   EQU       *
         PROGSTOP
            .
            .
            .
         ENDPROG
         END
```

STIMER - Set a system timer

The STIMER instruction sets the system timer for the number of seconds or milliseconds that you specify. You can use the instruction to:

- Delay program execution
- Post an event control block (ECB) in your program after a certain interval has elapsed
- Produce a return code after a certain interval has elapsed.

To avoid unnecessary program delays, you can code the STIMER instruction before instructions that request input, such as READTEXT or GETVALUE. When the instruction prompts an operator for data, the STIMER instruction gives the operator a specific amount of time to respond. If the operator does not respond to the prompt within the interval you specify, your program can continue processing. The STIMER instruction also prevents a program from tying up a terminal indefinitely while waiting for a response.

Syntax:

```
label        STIMER   count,action,SECS,P1=,P2=
                or
label        STIMER   RESET

Required:    count or RESET
Defaults:    count in milliseconds
Indexable:   count
```

Operand ***Description***

count A positive integer or the label of a positive integer (a word value) that specifies the timer setting in milliseconds or seconds.

The minimum timer setting is either 1 millisecond or second. The maximum setting is either 65,535 milliseconds or seconds.

Note: When using a model 4952, 4954 or 4956 processor, the minimum setting should not be less than 3 milliseconds.

action Specifies how the system timer operates. You can code one of three options: WAIT, TIO or ecbad. If you omit this operand, you must code a comma in its place to show that you have left the positional operand blank. In addition, if you do not code one of the three options, you must code a subsequent WAIT instruction with the keyword TIMER specified as the event for which you are waiting.

STIMER - Set a system timer *(continued)*

The timer options are as follows:

WAIT Suspends program execution until the interval you specified on the count operand has expired.

TIO Provides a return code of -5 in the task control block of the task containing the STIMER instruction when the interval you specified on count operand has expired. The first word of the task control block will contain the return code.

Use this option when you want to set a time limit on an instruction that requests operator input.

ecbad Code the label of an event control block (ECB) that the system posts when the interval you specified on the count operand has expired. The system places a value of -5 in the ECB.

Note: If the ECB to be posted is in another partition, you must move the address space of the ECB into $TCBADS before executing the STIMER instruction. The address space is equal to the partition number minus 1. An ECB in partition 2, for example, is in address space 1.

SECS Specifies that the value of the count operand is in seconds rather than milliseconds.

RESET Cancels the timer if the event the program is waiting for occurs before the interval on the timer has expired. You must code STIMER with RESET when you have specified TIO or ecbad on a previous STIMER instruction.

When you specify TIO, code an STIMER with RESET following the instruction that has the time limit on it. When you specify ecbad, code an STIMER with RESET following the WAIT instruction that waits for the ECB to be posted. Both uses of the RESET operand are shown in the coding examples for this instruction.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

STIMER - Set a system timer *(continued)*

Special Considerations

The following are some special considerations to keep in mind when you code the STIMER instruction:

- If you code an error exit routine that your program can invoke while a timer is set, you must reset the timer in your routine.
- Two STIMER instructions without an intervening WAIT will cause the interval specified by the first STIMER instruction to be replaced by the interval specified by the second STIMER instruction.
- With a 2741 terminal, if you use the TIO option of STIMER to set a timer for an instruction that requests input (for example, a READTEXT), normal program execution can be affected if the interval on the timer is allowed to expire. When the timer expires, the 2741 will be in a transmit state. For this reason, the device will be unable to do any output operations, such as a PRINTEXT. In this case, your program must reissue the instruction that requested input and an operator must respond to it by pressing the attention or RETURN key.

Syntax Examples

1) The STIMER instruction starts a 20-second timer. The WAIT instruction suspends task execution until the 20-second interval has elapsed. The WAIT instruction is required because the STIMER instruction does not specify one of the timer options.

```
S1       STIMER   20,,SECS
            .
            .
            .
         WAIT     TIMER
```

2) The STIMER instruction sets a timer for 30,000 milliseconds. Execution does not resume until after that interval has elapsed.

```
S2       STIMER   30000,WAIT
```

3) The MOVE instruction moves a value of 100 into SECONDS. The parameter naming operand on the STIMER instruction, P1=, receives the value for the count operand. The STIMER instruction halts task execution for 100 seconds, then passes control to the instruction following the S3 label.

```
         MOVE     SECONDS,100
S3       STIMER   0,WAIT,SECS,P1=SECONDS
```

STIMER - Set a system timer *(continued)*

Coding Examples

1) In the following example, the STIMER instruction at label S1 sets a timer for 120 seconds. If the operator does not enter his name within that period, the system places a return code of -5 in the task control block of the task. If the operator enters his name within the time limit, the STIMER with RESET following the READTEXT instruction cancels the portion of time remaining on the timer.

```
           .
           .
           .
        ENQT
S1      STIMER      120,TIO,SECS
        READTEXT    INPUT,'ENTER YOUR NAME',SKIP=1
        STIMER      RESET
        DEQT
           .
           .
           .
```

2) In this example, the STIMER instruction at the label TIME sets a timer for 60 seconds. Because the instruction contains the label of the event control block TIMEOUT, the system will post TIMEOUT if the 60-second interval expires before an event occurs. The STIMER with RESET following the WAIT instruction will cancel any time remaining on the timer if the system posts the ECB being waited on before the 60 seconds have elapsed.

```
           .
         RESET     TIMEOUT
           .
           .
           .
TIME     STIMER    60,TIMEOUT,SECS
           .
           .
           .
         WAIT      TIMEOUT
         STIMER    RESET
           .
           .
           .
         PROGSTOP
TIMEOUT  ECB
```

STIMER - Set a system timer *(continued)*

3) The STIMER instruction at label TIME1, in the following example, sets a timer for 180 seconds. When the interval expires, the system will post ECB1 unless the ECB is posted before that event. If the ECB is posted before the interval expires, the STIMER instruction at TIME2 prevents the system from posting the ECB again.

```
           .
         RESET     ECB1
         MOVE      TIME,180
         MOVEA     ECBADDR,ECB1
TIME1    STIMER    0,*,SECS,P1=TIME,P2=ECBADDR
         WAIT      ECB1
TIME2    STIMER    RESET
         IF        (ECB1,EQ,-5),GOTO,TIMEOUT
           .
           .
TIMEOUT  PRINTEXT 'TIMER HAS EXPIRED'
         PROGSTOP
ECB1     ECB
```

4) In the following example, the STIMER instruction at label SET sets a timer for 600 milliseconds. If the operator does not respond to the prompt message within the time interval, the system places a return of -5 in the first word of the task control block (TCB). The STIMER instruction at label RESET cancels any remaining time on the timer if the operator responds to the prompt message within 600 milliseconds. The IF instruction tests the return code to see if the interval has expired.

```
DATA     PROGRAM   START
           .
           .
SET      STIMER    600,TIO
         READTEXT HOLD,'ENTER YOUR WEIGHT',SKIP=1
RESET    STIMER    RESET
         IF        (DATA,EQ,-5),GOTO,TIMEOUT
           .
           .
TIMEOUT  PRINTEXT 'TIMER HAS EXPIRED'
```

Return Code

The return code is returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-5	Interval has expired

STORBLK - Define mapped and unmapped storage areas

The STORBLK statement defines the size and number of the storage areas your program can obtain with the GETSTG instruction. The SWAP instruction uses the mapped storage area which you define with this statement to gain access to the unmapped storage areas that you define.

Note: "Mapped storage" is the physical storage you defined on the SYSTEM statement during system generation. "Unmapped storage" is any physical storage that you did not include on the SYSTEM statement.

The STORBLK statement also creates a storage control block that:

- Contains the address of the mapped storage area your program acquires with GETSTG.
- Contains the location of and entries for the unmapped storage areas your program acquires with the GETSTG instruction.
- Records which unmapped storage area your program is using.

Your program can refer to the various fields in the storage control block by using the equates contained in the STOREQU module. To use these equates, code

```
         COPY     STOREQU
```

in your program. The STOREQU equates that may be of most use to you when coding your program are shown following the instruction operands.

The system releases the mapped and unmapped storage areas you defined with a STORBLK statement if the program containing the statement issues a PROGSTOP, if a program check occurs, or if you cancel the program with the $C command. You can also release storage areas with the FREESTG instruction.

Syntax:

```
label          STORBLK   TWOKBLK=,MAX=,EXT=

Required:      label,TWOKBLK=,MAX=
Defaults:      EXT=(points to an address in the storage control block)
Indexable:     none
```

STORBLK - Define mapped and unmapped storage areas *(continued)*

Operand	*Description*
TWOKBLK=	The size of the mapped storage area in 2K-byte blocks. Each 2K-byte block is equal to 2048 bytes of storage. Code a positive integer. The unmapped storage areas you define with the MAX= operand will also be this size. The maximum value you can specify for this operand is 32.
MAX=	The number of unmapped storage areas your program requires. The GETSTG instruction obtains these unmapped storage areas for your program.
EXT=	The label of an optional area outside the storage control block where the values that point to the unmapped storage areas can reside. The word size of this area must be equal to twice the value of the TWOKBLK parameter times the MAX parameter. For example, if you specify TWOKBLK=2 and MAX=8, the extension area would have to be 32 words long. You must initialize each word of the extension area to -1 (X'FFFF') If you do not code this operand, the STORBLK statement generates an area to store the values that point to the unmapped storage areas that your program obtains.

STOREQU Equates

You may find the following equates helpful when coding a program that uses unmapped storage:

$STORMAP   Address of the mapped storage area.

$STORMPK   Address space of the mapped storage area (partition number minus one).

Syntax Examples

1) Defines a mapped storage area of 40K bytes and two unmapped storage areas of 40K bytes each.

```
BLOCK       STORBLK     TWOKBLK=20,MAX=2
```

2) Defines a mapped storage area of 20K bytes and four unmapped storage areas of 20K bytes each.

```
BLOCK1      STORBLK     TWOKBLK=10,MAX=4
```

STORBLK - Define mapped and unmapped storage areas *(continued)*

3) Defines a mapped storage area of 4K bytes and eight unmapped storage areas of 4K bytes each. The values that point to these unmapped storage areas reside in A. Note that the extension area is 32 words long because your program specifies TWOKBLK=2 and MAX=8. You must initialize the extension area to '-1'.

```
BLOCK2      STORBLK     TWOKBLK=2,MAX=8,EXT=A
A           DC          32F'-1'
```

4) Defines a mapped storage area of 2K bytes and 20 unmapped storage areas of 2K bytes each. The values that point to these unmapped storage areas reside in HOLD.

```
BLOCK2      STORBLK     TWOKBLK=1,MAX=20,EXT=HOLD
HOLD        DC          40F'-1'
```

Coding example

See the SWAP instruction for a coding example that contains the STORBLK statement.

SUBROUT - Define a subroutine

The SUBROUT statement defines a callable subroutine. You can pass up to five parameters, or arguments, to the subroutine. The subroutine must include a RETURN instruction to provide linkage back to the calling task. Nested subroutines are allowed, and a maximum of 99 subroutines are permitted in each 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.

You can call a subroutine from more than one task. When called, the subroutine executes as part of the calling task. Because subroutines are not reentrant, you should ensure serial use of the subroutine with the ENQ and DEQ instructions.

Note: Do not code a TASK statement within a subroutine.

Syntax:

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

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

Operand	*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 complete 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 statement. These parameters are defined automatically as single-precision values. For instance, assume you have 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.

SUBROUT - Define a subroutine *(continued)*

Coding Example

The CALL instruction in this example calls the subroutine named CHKBUFF. The calling program passes two parameters to the CHKBUFF subroutine. The first parameter, BUFFLEN, is a variable containing the maximum allowable buffer count. The second parameter, BUFFEND, is the address of the next instruction to be executed if the buffer is full.

```
         CALL      CHKBUFF,BLEN,BEND
          .
          .
          .
         SUBROUT   CHKBUFF,BUFFLEN,BUFFEND
*
         SUBTRACT BUFFLEN,1
         IF        (BUFFLEN,GE,MAX)
            GOTO   (BUFFEND)
         ENDIF
         ADD       BUFFLEN,1
         RETURN
*
*
MAX      DATA      F'256'
```

SUBTRACT - Subtract integer values

The SUBTRACT instruction subtracts an integer value in operand 2 from an integer value in operand 1. The values can be positive or negative. (See the DATA/DC statement for a description of the various ways you can represent integer data.) To subtract floating-point values, use the FSUB instruction.

You can abbreviate this instruction as SUB.

EDX does not indicate an overflow condition for this instruction.

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
```

Operand	*Description*
opnd1	The label of the data area from which opnd2 is subtracted. Opnd1 cannot be a self-defining term. The system stores the result of the SUBTRACT operation in opnd1 unless you code the RESULT operand.
opnd2	The value subtracted from opnd1. You can specify a self-defining term or the label of a data area. The value of opnd2 does not change during the operation.
count	The number of consecutive values in opnd1 on which the operation is to be performed. The maximum value allowed is 32767.
RESULT=	The label of a data area or vector in which the result is placed. Opnd1 is not changed if you specify RESULT. This operand is optional.
PREC=xyz	Specify the precision of the operation in the form xyz, where x is the precision for opnd1, y is the precision for opnd2, and z is the precision of the result ("Mixed-Precision Operations" on page LR-436 shows the precision combinations allowed for the SUBTRACT instruction). You can specify single-precision (S) or double-precision (D) for each operand. Single-precision is one word in length; double-precision is two words in length. The default for opnd1, opnd2, and the result is single-precision. If you code a single letter for PREC, the letter applies to opnd1 and the result. Opnd2 defaults to single precision. If, for example, you code PREC=D, opnd1 and the result are double-precision and opnd2 defaults to single-precision.

SUBTRACT - Subtract integer values *(continued)*

If you code two letters for PREC, the first letter applies to opnd1 and the result, and the second letter applies to opnd2. With PREC=DD, for example, opnd1 and the result are double-precision and opnd2 is double-precision.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Mixed-Precision Operations

The following table lists the precision combinations allowed for the SUBTRACT instruction:

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

Syntax Examples

1) Subtract 2 from 5 and place the result of the operation in C.

```
         SUB    A,B,RESULT=C     SINGLE-PRECISION SUBTRACT
          .
          .
         PROGSTOP
A        DATA   F'5'
B        DATA   F'2'
C        DATA   F'0'
          .
          .
```

2) Subtract the value at the address defined by 2 plus the contents of #2 from the value in data area A. Replace the contents of A with the results of the operation.

```
          .
          .
         SUB    A,(2,#2)    SUBTRACT DATA AT (2,#2) FROM A
          .
          .
         PROGSTOP
A        DATA   F'10'
          .
          .
```

SWAP - Gain access to an unmapped storage area

The SWAP instruction gains access to an unmapped storage area you obtained with the GETSTG instruction. Your program gives up the use of a block of mapped storage you obtained with GETSTG to gain access to one or more blocks of unmapped storage.

Note: "Mapped storage" is the physical storage you defined on the SYSTEM statement during system generation. "Unmapped storage" is any physical storage that you did not include on the SYSTEM statement.

Refer to *Event Driven Executive Language Programming Guide* for more information on how to code programs that use unmapped storage.

Syntax:

```
label        SWAP     name,number,ERROR=,P1=,P2=

Required:    name
Defaults:    value of 0 for number
Indexable:   none
```

Operand	*Description*
name	The label of a STORBLK statement that defines the mapped and unmapped storage areas this instruction uses.
number	The number of the unmapped storage area that you want to use. Your program has access to this area until it issues another SWAP instruction. The number must be between 0 and the maximum number of unmapped storage areas you defined on the STORBLK statement. You can code a positive integer or the label of a positive integer. By coding 0 for this operand, your programs regains access to the mapped storage area. It is your responsibility to keep track of the contents of each unmapped storage area.
ERROR=	The label of the first instruction of the routine to receive control if an error condition occurs while this instruction is executing.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

SWAP - Gain access to an unmapped storage area *(continued)*

Syntax Examples

1) Get access to the second unmapped storage area defined in the STORBLK statement, BLOCK.

```
         SWAP     BLOCK,2
```

2) Get access to the fourth unmapped storage area defined in the STORBLK statement, BLOCK.

```
         SWAP     BLOCK,A
           .
           .
           .
A        DATA     F'4'
```

Coding Example

The following program reads payroll data into three unmapped storage areas, updates the data, and writes the data back to a disk data set. The program begins by acquiring a mapped storage area of 2K bytes and three unmapped storage areas of 2K bytes apiece. The STORBLK statement at label A defines the size of the mapped storage area and the number of unmapped storage areas to be acquired.

The MOVE instruction at label M1 moves the address of the mapped storage area into register 1. The MOVE instruction uses the STOREQU equate $STORMAP to find the address. The MOVE instruction at label M2 moves the number of the first unmapped storage area the program uses into the COUNT field. The DO loop beginning at label LOOP1 executes a SWAP instruction that gives up access to the mapped storage area and uses its segmentation register to get access to the first unmapped storage area. The READ instruction reads 8 records into the first unmapped storage area. The program updates the COUNT field and reads 8 records into the next unmapped storage area.

When the program reads the payroll records into each of the unmapped storage areas, the COUNT field is reset to 1, and the loop at label LOOP2 begins. This DO loop moves the data in PAYCODE into the PAYCODE field of each record in the unmapped storage area. The WRITE instruction then writes the records back to the disk data set. The loop continues until the program has updated the records in each unmapped storage area.

The FREESTG instruction releases the mapped and unmapped storage areas acquired with the GETSTG instruction. This instruction also restores the segmentation register values for the mapped storage area.

SWAP - Gain access to an unmapped storage area *(continued)*

```
PAYROLL   PROGRAM     START,DS=(PAYROLL)  GET MAPPED AND UNMAPPED AREAS
START     EQU         *                   GET MAPPED AREA ADDRESS FROM
          GETSTG      A,TYPE=ALL          STORAGE CONTROL BLOCK
M1        MOVE        #1,A+$STORMAP       FIRST UMMAPPED AREA
*
M2        MOVE        COUNT,1
*
LOOP1     DO          3                   FOR EACH UNMAPPED AREA
SWAP1       SWAP        A,COUNT           SUBSTITUTE UNMAPPED AREA
            READ        DS1,(0,#1),8      READ IN DATA FROM DISK
            ADD         COUNT,1           GET NEXT UNMAPPED AREA
          ENDDO
          MOVE        COUNT,1             FIRST UNMAPPED AREA
LOOP2     DO          3                   FOR EACH UNMAPPED AREA
SWAP2       SWAP       A,COUNT
LOOP3       DO         8                  FOR RECORDS IN UNMAPPED AREA
              MOVE       (+PYCD,#1),PAYCODE     UPDATE PAYCODE
              ADD        #1,256           NEXT RECORD
            ENDDO
            MOVE       #1,A+$STORMAP      GET MAPPED AREA ADDRESS
            WRITE      DS1,(0,#1),8       WRITE BACK TO DISK
           ADD        COUNT,1             GET NEXT UNMAPPED AREA
         ENDDO
         FREESTG     A,TYPE=ALL
         PROGSTOP
A        STORBLK     TWOKBLK=1,MAX=3
COUNT    DATA        F'0'
PAYCOD   DATA        F'0'
PYCD     EQU         10                   PAYCODE FIELD IS 10 BYTES
*                                         INTO RECORD
         COPY        STOREQU
         ENDPROG
          END
```

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Successful completion
1	The number of the unmapped storage area you request is beyond the number of areas defined on the STORBLK statement
2	SWAP area is not initialized
100	No unmapped storage support in the system

TASK - Define a program task

The TASK statement defines a task that executes asynchronously with the task that starts or "attaches" it. The system executes tasks according to their priority. Use the PROGRAM statement to define the primary task or main program.

Each task in a program, except the primary task, begins with a TASK statement and must end with an ENDTASK statement.

If you want to link-edit your program, place all TASKS you wish to attach using the ATTACH instruction in the same module. The assembler will only chain TASKS within the module it assembles. Your application program will have to chain the TASKS together if they are not within the same module. Modify the correct field in the TCB to chain tasks across modules.

Code TASK statements only within main programs, not within subprograms (MAIN=NO on the PROGRAM statement).

Syntax:

```
taskname     TASK     start,priority,EVENT=,TERMERR=,FLOAT=,
                      ERRXIT=

Required:    taskname,start
Defaults:    priority=150,FLOAT=NO
Indexable:   none
```

Operand	*Description*
taskname	The label you assign to the task. The system generates a control block for each task in the program. Refer to this control block as the task control block (TCB). The system generates the TCB when it encounters an ENDPROG statement. The label of the task's TCB is the label you specify with this operand. The supervisor uses the TCB to store instruction return codes. By referring to the TCB (the taskname) in your program, you can determine if an operation completed successfully.
start	The label of the first instruction you want the system to execute when the task first attaches.
priority	The priority you assign to the task. The range is from 1 (highest priority) to 510 (lowest priority). Tasks with priorities 1-255 run on hardware interrupt level 2 and those with 256-510 run on hardware interrupt level 3.

TASK - Define a program task *(continued)*

Priorities rank tasks according to their realtime 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 as complete at the end of this task. The attaching task can, if desired, synchronize its operation by issuing a WAIT for this event. Do not define this event name explicitly by an ECB since your system generates it automatically.

TERMERR= The label of the routine to receive control if an unrecoverable terminal I/O error occurs.

If such an error occurs, the first word of the task control block (TCB) contains the return code indicating the error. The second word of the TCB contains the address of the instruction that was executing when the error occurred. If you do not code TERMERR, the return code is available in the task code word. You should use TERMERR for detecting errors because the task code word is subject to modification by numerous system functions. Therefore, It may not always reflect the true status of terminal I/O operations.

FLOAT= YES, if this task uses floating-point instructions.

NO (the default), if this task does not use floating-point instructions.

ERRXIT= Specifies the label of a three-word list. That list points to a routine which is to receive control if a hardware error or program exception occurs while this task is executing. Prepare the task error exit routine to handle any type of program or machine error completely. See the *Event Driven Executive Language Programming Guide* for additional information on the use of task error exit routines. It is often necessary to release resources even though your program cannot continue because of an error. This is the case if the primary task is part of a program which shares resources with other programs. These resources may be, for example, QCBs, ECBs, or Indexed Access Method update records. The supervisor does not release resources for you, but the task error exit facility allows you to take whatever action 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. Two types of informational code are placed here from the point where an error occurred before the exit routine is entered. These are the Level Status Block (LSB) and the Processor Status Word (PSW). Refer to a Series/1 processor description manual for a description of the LSB and PSW.

TASK - Define a program task *(continued)*

A default task error exit routine is available to aid in problem diagnosis and correction. (Refer to *Event Driven Executive Language Programming Guide* for a detailed description of this routine and how to use it in your application program.)

Coding Example

The following example shows the use of the TASK statement in a program with multiple tasks. The program reads a record from the data set MYFILE and prints the first 8 bytes of that record. The program begins by attaching TASK1. TASK1 is the label of a TASK statement. TASK1 prints the message at label P1 and reads a record from MYFILE into the buffer BUF. The MOVE instruction moves the first 8 bytes of BUF into the text buffer labeled REC. When TASK1 ends, it signals the event by posting the ECB at label ECB1.

The main program attaches the task at label TASK2. The WAIT instruction at label W1 checks ECB1 to see if TASK1 has completed. TASK2 then enqueues the printer and prints the contents of REC. When TASK2 ends, it posts the event specified on the EVENT= operand of the TASK statement. The main program receives control and the WAIT instruction at label W2 checks to see if TASK2 has ended. The PRINTEXT instruction at label P4 prints the message "PROGRAM COMPLETE" and the program ends.

```
READTASK     PROGRAM     START,DS=((MYFILE,EDX40))
START        EQU         *
             ATTACH      TASK1
             ATTACH      TASK2
W2           WAIT        EVENT
P4           PRINTEXT    'PROGRAM COMPLETE',SKIP=2
             PROGSTOP
ECB1         ECB
BUF          BUFFER      256,BYTES
REC          TEXT        LENGTH=8
*************************************************
TASK1        TASK        NEXT
NEXT         ENQT        $SYSPRTR
P1           PRINTEXT    '@TASK1 ATTACHED'
             READ        DS1,BUF,1
             MOVE        REC,BUF,(8,BYTES)
             POST        ECB1
             DEQT        $SYSPRTR
             ENDTASK
*************************************************
TASK2        TASK        W2,EVENT=EVENT
W1           WAIT        ECB1
             ENQT        $SYSPRTR
P2           PRINTEXT    '@TASK2 ATTACHED',SKIP=1
P3           PRINTEXT    REC,SKIP=1
             DEQT        $SYSPRTR
             ENDTASK
*************************************************
             ENDPROG
             END
```

TCBGET - Get task control block data

The TCBGET instruction obtains data from a specified field in the task control block (TCB) of the currently executing task.

Syntax:

```
label        TCBGET    opnd1,opnd2,P1=

Required:    opnd1
Defaults:    $TCBVER (opnd2)
Indexable:   opnd1
```

Operand	***Description***
opnd1	The label of a one-word data area where the system stores the specified TCB field.
opnd2	This operand determines which TCB field the system will copy. If you do not code this operand, the default $TCBVER will be used. $TCBVER contains the address of the current TCB.

Code this operand using any of the TCB equate names. Some examples are:

$TCBCO - first word of the TCB

$TCBCO2 - second word of the TCB

$TCBADS - current address key

$TCBVER - address of the current TCB

You will find a complete list of TCB equates in the *Internal Design*.

Note: Spell entries for this operand as specified in the TCB equates. The EDX assembler may not flag some you spell incorrectly.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

TCBGET - Get task control block data *(continued)*

Syntax Examples

1) The following example does not include code for opnd2. Therefore, it defaults to $TCBVER. The system stores the contents of $TCBVER (current TCB address) at variable A.

```
LABEL1    TCBGET    A
            .
            .
            .
A         DATA      F'0'
```

2) In this example, the contents of the TCB field $TCBCO are stored in software register 1.

```
LABEL2    TCBGET    #1,$TCBCO
```

TCBPUT - Store data in a task control block

The TCBPUT instruction stores a value in the specified field of the task control block (TCB) of the currently executing task.

Syntax:

```
label        TCBPUT     opnd1,opnd2,P1=

Required:    opnd1
Defaults:    $TCBCO (opnd2)
Indexable:   opnd1
```

Operand	*Description*
opnd1	The TCB field opnd2 points to and the data your system stores in opnd1. You can specify the label of a one-word data area containing the data to be stored or you can specify a self-defining term.
opnd2	This operand specifies which TCB field the system will modify. Use the following names and corresponding fields in opnd2: $TCBCO - first word of the TCB $TCBCO2 - second word of the TCB $TCBADS - current address key A complete list of TCB equates is in the *Internal Design*. **Note:** Spell entries for this operand as specified in the TCB equates. The EDX assembler may not flag some you spell incorrectly.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Syntax Examples

1) The following program example moves the value 7 into the first word of the TCB. It allows opnd2 to default to $TCBCO.

```
LABEL1    TCBPUT    +7
```

2) Your system adds 6 to the contents of the word at the address to which #2 points. It then stores the result in the $TCBADS field of the current TCB.

```
LABEL2    TCBPUT    (6,#2),$TCBADS
```

TERMCTRL - Request special terminal functions

The TERMCTRL instruction requests the execution of special terminal-control functions. The functions available with the TERMCTRL instruction vary depending on the device you are using. The "TERMCTRL Functions Chart" shows the devices to which you can issue a TERMCTRL instruction, and the functions that you can select for these devices. You will find the syntax of the TERMCTRL instruction for each of these devices following the chart.

The supervisor places a return code in the first word of the task control block (taskname) whenever a TERMCTRL instruction causes a terminal I/O operation to occur. If the return code is not a -1, your system places the address of this instruction in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this book and also in the *Messages and Codes*.

TERMCTRL Functions Chart

The chart on the following pages shows the devices to which you can issue a TERMCTRL instruction, and the various functions available with each device. The device names appear across the top of the chart and the functions for these devices are listed down the left side of the chart. The 4975 terminal device described on this chart is not the 4975-01A ASCII Printer. The 4975-01A ASCII Printer uses data streams and not TERMCTRL statements to control printer operations. ("Request Special Terminal Function (4975-01A)" on page LR-334 explains data streaming on the 4975-01A ASCII Printer.)

TERMCTRL - Request special terminal functions *(continued)*

FUNCTIONS	DEVICE TYPES 2741	3101C	3101B	4013	4973	4974	4975	4980/ 4978	4979
BLINK cursor								X	
BLINK field			X						
UNBLINK								X	
BLANK screen			X					X	X
BLANK field			X						
DISPLAY	X	X	X	X	X	X	X	X	X
ENABLE									
ENABLEA									
ENABLET									
ENABLEAT									
DISABLE									
GETSTORE						X		X	
HIGH/LOW intensity			X						
LOCK/UNLOCK keyboard			X					X	X
PF									
PUTSTORE						X		X	
RING									
RINGT									
SET attention		X	X	X				X	X
SET lines per inch					X	X	X		
TONE			X					X	
Special functions							X		

Note: Device 3101B refers to a 3101 in block mode. Device 3101C refers to a 3101 in character mode. Device 4975 does not refer to the 4975-01R or the 4975-01A ASCII printer. (See "Request Special Terminal Function (4975-01A)" on page LR-334 for information about data streaming on the 4975-01A ASCII printer.)

TERMCTRL - Request special terminal functions *(continued)*

FUNCTIONS	DEVICE TYPES 5219/ 5224	5225	ACCA/ MODEM	ACCA	TTY	VIRT	GPIB	S1/S1
BLINK cursor								
BLINK field								
UNBLINK								
BLANK screen								
BLANK field								
DISPLAY	X	X			X	X	X	
ENABLE			X					
ENABLEA			X					
ENABLET			X					
ENABLEAT			X					
DISABLE			X					
GETSTORE								
HIGH/LOW intensity								
LOCK/UNLOCK keyboard								
PF						X		
PUTSTORE								
RING			X					
RINGT			X					
SET attention			X	X	X	X		
SET lines per inch	X	X						
TONE								
Special functions	X	X					X	X

Note: ACCA and ACCA with MODEM are listed as devices in this chart.

TERMCTRL - Request special terminal functions *(continued)*

2741 Communications Terminal

Syntax:

label	TERMCTRL DISPLAY
Required:	DISPLAY
Defaults:	none
Indexable:	none

Operand	*Description*
DISPLAY	Causes any buffered output to be written to the 2741.

Coding Example

The following example displays the contents of the buffer on a 2741 terminal.

```
         TERMCTRL   DISPLAY         DISPLAY BUFFER
```

TERMCTRL - Request special terminal functions *(continued)*

3101 Display Terminal (Block Mode)

A 3101 in block mode uses an attribute byte at the beginning of a data field. The attribute byte defines the characters of a field as protected, unprotected, modified, or not modified. The attribute byte also defines the display mode as high intensity, low intensity, blinking, or nondisplay. The field extends up to the next attribute byte or the end-of-screen, whichever occurs first. The attribute byte appears as a protected blank on the screen.

In general an I/O operation directed to a 3101 in block mode, results in a 3101 data stream being transferred between the 3101 and processor storage. The 3101 data stream consists of escape sequences, attribute characters, and data. With input, the 3101 transfers a 3101 data stream into processor storage. With output, a 3101 data stream must be built in processor storage to be transferred to the 3101. The 3101 interprets the escape sequences as control commands. The attribute bytes appear on the screen as protected blanks and the data is displayed on the screen in a manner controlled by the attribute bytes.

Terminal I/O allows you to write messages in any display mode to a 3101 in block mode. The 3101 block mode support conditionally inserts the correct attribute bytes in the 3101 data stream for you before the write operation.

For a roll screen read operation, terminal I/O also allows you to enter data in any display mode. The 3101 block mode support places the correct attribute byte at the beginning of the input field. The data you enter takes on the display mode defined by the attribute byte.

You set the display mode for input and output operations with the ATTR operand. You must code the SET function with the ATTR operand (SET,ATTR=). Do not include other operands in the instruction when you are establishing the attribute byte. Once set from a program with TERMCTRL SET,ATTR= instruction, the attribute byte set will remain in effect. There are two ways to change it for the 3101 terminal in block mode. One is to issue another TERMCTRL SET,ATTR= instruction from an application program. The other is to request a new attribute byte for the terminal with the $TERMUT1 utility.

When you code STREAM=YES, the system ignores the attribute byte you specified with the ATTR operand. Neither the system nor a DEQT or PROGSTOP instruction resets the attribute byte in this case. The attribute byte remains set even after the program has ended.

The STREAM operand gives you control over whether terminal I/O will remove or insert 3101 special characters during input or output operations. You must code the SET function with the STREAM operand (SET,STREAM=). Once a program issues the TERMCTRL SET,STREAM= instruction to a 3101 in block mode, it remains in effect until the program issues another TERMCTRL SET,STREAM= instruction to the terminal or until you change the STREAM option with the $TERMUT1 utility. A DEQT or PROGSTOP instruction does not reset the option you select with the STREAM operand and it remains in effect even after the program has ended.

The ACCA TERMCTRL functions are also applicable to a 3101 in block mode. For a description of those functions see "ACCA Attached Devices" on page LR-483.

TERMCTRL - Request special terminal functions *(continued)*

Syntax:

```
label          TERMCTRL  function,ATTN=,ATTR=,STREAM=

Required:      function
Defaults:STREAM=NO
Indexable:none
```

Operand	***Description***	
function:		
	TONE	Causes the 3101 in block mode to sound the audible alarm.
	DISPLAY	Causes the system to write to the device any buffered output. In addition, for 3101 block mode, the cursor position is updated accordingly.
	LOCK	Locks the keyboard for a 3101 in block mode.
	UNLOCK	Unlocks the keyboard for a 3101 in block mode.
	SET	The action of the SET function for a 3101 in block mode depends on how you code the ATTN=, ATTR=, and STREAM= operands.

ATTN= YES, to enable the attention and PF key functions.

NO, to disable the attention and PF key functions.

ATTR= HIGH (the default), for a display mode of high intensity for both input and output.

LOW, for a display mode of low or normal intensity for both input and output.

BLINK, causes a blinking display for both input and output.

BLANK, prevents the display of input or output characters. This mode is useful for reading data, such as a password, that should not be displayed on the screen. Change this option when you no longer require it. The terminal is unable to display data while ATTR=BLANK is in effect.

NO, for output, specifies that no attribute byte is to be placed in the data stream. For input, the attribute byte depends on the current TERMCTRL SET,ATTR= in effect. If a SET,ATTR= has not been issued, the system uses the default, ATTR=HIGH.

TERMCTRL - Request special terminal functions *(continued)*

YES, clears a previous TERMCTRL SET,ATTR=NO instruction. This operand has no effect if the previous TERMCTRL SET,ATTR= instruction does not contain ATTR=NO.

For a roll screen read operation, terminal I/O also allows you to enter data in any display mode. The 3101 block mode support places the correct attribute byte at the beginning of the input field. The data you enter takes on the display mode defined by the attribute byte.

STREAM= YES, for output operations, shows that you have already supplied in the text or buffer area the attribute bytes and escape sequences the terminal needs to do an output operation. For input operations, it allows you to receive the entire 3101 data stream in processor storage exactly as it is transmitted by the device.

Note: Certain terminal I/O instructions, such as GETEDIT, GETVALUE, and QUESTION, are not recommended for use with STREAM=YES. You also should be familiar with the 3101 device and terminal I/O internals to use this option effectively.

NO, for output operations, shows that the system should insert the required escape sequences and attribute bytes in the text or buffer area before displaying data on the 3101 screen.

For input operations, it allows the system to remove 3101 special characters from the 3101 data stream before returning control to your program.

The default is STREAM=NO.

If you code STREAM=YES in your application program, issue a TERMCTRL SET,STREAM=NO before a PROGSTOP or DETACH instruction to restore the default.

For either YES or NO, conversion to and from EBCDIC takes place for both input and output. The only exception to this occurs when you code XLATE=NO on a READTEXT or PRINTEXT instruction. Then, for the duration of that instruction, the system ignores the STREAM option you coded and no EBCDIC conversion takes place, nor does the system insert or remove any 3101 special characters.

TERMCTRL - Request special terminal functions *(continued)*

4013 Graphics Terminal

Syntax:

```
label         TERMCTRL  function,ATTN=

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

Operand	*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 the system to write to the 4013 any buffered output.
ATTN=	NO, to disable the attention function.	
	YES, to enable the attention function.	
	This operand is required when function is SET.	

Coding Example

The following example displays the contents of the buffer on a 4013 terminal. The program then disables the attention key and loads an application program named PAYROLL. When the PAYROLL program returns control to the loading program, the instructions at ENABLE1, enables the attention key before the program stops.

```
          TERMCTRL  DISPLAY         DISPLAY BUFFER
DISABLE1  TERMCTRL  SET,ATTN=NO     DISABLE ATTENTION FUNCTION
          LOAD      PAYROLL,DS=(EMPFILE,ADDRFILE)
            .
            .
            .
ENABLE1   TERMCTRL  SET,ATTN=YES    ENABLE ATTENTION FUNCTION
          PROGSTOP
```

TERMCTRL - Request special terminal functions *(continued)*

4973 Printer

Syntax:

```
label          TERMCTRL  function,LPI=,DCB=

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

Operand	***Description***
function:	
	SET — Sets the number of lines per inch and causes any buffered output to be printed. The system also resets the current output position to the beginning of the left margin. When you specify SET, you must also specify LPI.
	DISPLAY — Causes the system to write to the 4973 any buffered output.
LPI=	The number of lines per inch (either 6 or 8) the 4973 is to print. This operand is required when the SET function is specified.
DCB=	The label of an 8-word device control block you define with the DCB statement. The 4973 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device. If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions. Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4973 hardware and terminal I/O internals when you use this operand.

TERMCTRL - Request special terminal functions *(continued)*

Syntax Examples

1) Print the contents of the buffer.

```
WRITEPTR  TERMCTRL  DISPLAY
```

2) Set printer to print eight lines per inch.

```
          TERMCTRL  SET,LPI=8
```

3) Set printer to print six lines per inch.

```
          TERMCTRL  SET,LPI=6
```

TERMCTRL - Request special terminal functions *(continued)*

4974 Printer

Syntax:

```
label        TERMCTRL  function,opnd1,opnd2,count,TYPE=,LPI=,
                       DCB=

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

Operand ***Description***

function:

SET — Sets the number of lines per inch and causes any buffered output to be printed. The system also resets the current output position to the beginning of the left margin.

When you specify SET, you must also specify LPI.

DISPLAY — Causes the system to write to the 4974 any buffered output.

PUTSTORE — Transfers control data from the processor to the 4974 wire image buffer. If PUTSTORE is specified, operands opnd1, opnd2, count, and TYPE are required.

GETSTORE — Transfers control data from the 4974 wire image buffer to the processor. If GETSTORE is specified, opnd1, opnd2, count, and TYPE are required.

opnd1 The address in the processor from which or to which the information is to be transferred. Required with function PUTSTORE or GETSTORE.

opnd2 The address in the 4974 wire image buffer to which or from which the information is to be transferred. Required with function PUTSTORE or GETSTORE.

count The number of bytes to be transferred. Required with function PUTSTORE or GETSTORE.

TERMCTRL - Request special terminal functions *(continued)*

TYPE= The type of PUTSTORE or GETSTORE 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 show 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 shows replacement (1) or nonreplacement (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.

LPI= The number of lines per inch, either 6 or 8, the 4974 is to use for printing. This operand is required when the SET function is coded.

DCB= The label of an 8-word device control block you define with the DCB statement. The 4974 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system performs a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4974 hardware and terminal I/O internals when you use this operand.

TERMCTRL - Request special terminal functions *(continued)*

Coding Examples

1) This example initializes the 4974 wire image buffer to the standard EBCDIC character set. The example also replaces the standard dollar sign ($) with its alternate, the English sterling symbol (hex code 5B), and replaces the standard cent sign (¢) with its alternate, the dollar sign ($) (hex code 4A).

```
          ENQT      PTR1        ENQUEUE PRINTER
            .
            .
            .
          TERMCTRL  PUTSTORE,REPLACE,0,2,TYPE=2
            .
            .
            .
REPLACE   DATA      X'1200'
PTR1      IOCB      T4974
```

2) If RDWRFLAG in the following example equals zero, the TERMCTRL instruction transfers 768 bytes of control data from the processor to the 4974 wire image buffer. If the RDWRFLAG is not zero, the instruction transfers 768 bytes of control data from from the 4974 wire image buffer to the processor.

```
        ENQT        PTR1         ENQUEUE PRINTER
          .
          .
          .
        SUBROUT     SETPRNTR,RDWRFLAG
        IF          (RDWRFLAG,EQ,0)     IF WRITE WIRE IMAGE OPERATION
          TERMCTRL  PUTSTORE,BUFF,0,768,TYPE=1
          ELSE                          ELSE READ WIRE IMAGE BUFFER
          TERMCTRL  GETSTORE,BUFF,0,768,TYPE=1
        ENDIF
        RETURN
BUFF    DATA        768H'0'             BUFFER AREA FOR 4974 WIRE IMAGE
PTR1    IOCB        T4974
```

TERMCTRL - Request special terminal functions *(continued)*

4975 Printer

Syntax:

```
label         TERMCTRL  function,LPI= or print operand,DCB=

Required:     function
Defaults:     none
Indexable:    CHARSET,PDEN,PMODE
```

Operand	*Description*

function:

SET — Sets the number of lines per inch and causes any buffered output to be printed. The system also resets the current output position to the beginning of the left margin.

If you do not specify the the LPI operand, you must code the SET function along with one of four print operands that allow you to set and control the special print functions available with the 4975 Model 1 and Model 2 printers. (See "SET Function Operands" on page LR-460 for a description of each of the print operands.)

Note: You must code the SET function along with either the LPI operand or one of the print operands.

DISPLAY — Causes the system to write to the 4975 any buffered output. No operands are valid with this function.

LPI= The number of lines per inch (either 6 or 8) the 4975 is to print. Use this operand only with the SET function.

DCB= The label of an 8-word device control block you define with the DCB statement. The 4975 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4975 hardware and terminal I/O internals when you use this operand.

TERMCTRL - Request special terminal functions *(continued)*

SET Function Operands

The four SET function operands allow you to:

- Specify the print mode on a 4975 Model 2 printer (PMODE).
- Specify the density of printed characters (PDEN).
- Specify the language character set (CHARSET).
- Restore the default values for the printer (RESTORE).

You can code only one print operand on each TERMCTRL statement. When specifying parameters on the PMODE, PDEN, and CHARSET operands, you can code the parameter name, an indexed value, or an address. A given address must not have the same name as the allowable parameters.

To simplify the coding of addresses and indexed values, the system provides an equate table, EQU4975. The parameter equate is the parameter name preceded by a "$" sign. For example, the parameter equate for the Italian character set, ITAL, is $ITAL. Before using addresses or indexed values with the TERMCTRL statement, you must copy the equate module (EQU4975) into your application program with a COPY statement.

Note: To use the SET function operands, you must link-edit your program with $EDXLINK and specify an autocall to $AUTO,ASMLIB. Refer to the *Operator Commands and Utilities Reference* for details on the AUTOCALL option of $EDXLINK.

Operand	Description
PMODE=	Specifies the print mode to be used on a 4975 model 2 printer. **PMODE=DRAFT** — Print in draft-processing mode (all characters are equal in width). The 4975 Model 1 printer prints only in draft-processing mode. **PMODE=TEXT** — Print in text-processing mode with two passes of the print head (character width is variable). **PMODE=TEXT1** — Print in text-processing mode with a single pass of the print head. This option produces characters that do not have a full complement of dots. It can be used to check the format of printed output.
PDEN=	Specifies the density of printed characters on each line. You can select compressed, "normal," and expanded character density for the 4975 Model 2 printer. The 4975 Model 1 printer supports "normal" or expanded character density. If you code compressed for the 4975 Model 1 printer, the density defaults to expanded.

TERMCTRL - Request special terminal functions *(continued)*

In draft mode, the compressed density is 20 characters per inch, the "normal" density is 15 characters per inch, and the expanded density is 10 characters per inch.

In text mode (PMODE=TEXT or TEXT1), the size of individual characters varies (the letter "i", for example, is narrower than the letter "m"), and the number of characters per inch depends on the mix of characters in the data stream.

PDEN=NORM — Print in "normal" or typewriter-like characters. In draft mode, you can print up to 198 characters on a line.

PDEN=COMP — Print in compressed characters. In draft mode, you can print up to 230 characters on a line.

PDEN=EXPD — Print in expanded characters. In draft mode, you can print up to 132 characters on a line.

When you code the PDEN= operand, be sure the line length of your TEXT or BUFFER statement does not exceed the maximum line length for the density you choose.

CHARSET= Specifies the language character set to be used. The CHARSET operand changes the default character set specified during system generation. (See the TERMINAL statement for the 4975 printer in the *Installation and System Generation Guide.*)

The character set coded with the CHARSET operand becomes the new default for the printer. You can change the default character set with another TERMCTRL statement or with the $TERMUT1 utility. (See the *Operator Commands and Utilities Reference* for details on how to use the $TERMUT1 utility.)

The following character sets are available on the 4975 printer:

AUGE	Austrian and German
BELG	Belgian
BRZL	Brazilian
DNNR	Danish and Norwegian
FRAN	French
FRCA	French Canadian
INTL	International (multinational)
ITAL	Italian
JAEN	Japanese and English
KANA	Japanese Katakana

TERMCTRL - Request special terminal functions *(continued)*

PORT	Portugese
SPAN	Spanish (Spain)
SPNS	Spanish (other)
SWFI	Swedish and Finnish
UKIN	English (United Kingdom)
USCA	English (United States and Canada).

RESTORE Allows the printer to return to the default values previously defined in the TERMCTRL statement. These operands include PDEN, PMODE, CHARSET, and LPI. When altered, each causes a permanent change to the defaults established for the 4975 printer. The system restores the default values to those set with the last CT command of the $TERMUT1 utility or, if the CT command has not been used, to values specified at system generation.

When you change printer functions with a TERMCTRL statement, code the RESTORE option on another TERMCTRL statement to restore the original default values before your program ends.

Notes:

1. If any of the print operands are issued to devices other than the 4975, 5219, 5224 or 5224 printers, they will be ignored, and a return code of -1 will be returned to the issuing program.

2. Do not confuse the 4975-01A ASCII printer with the 4975 printer. The 4975-01A ASCII printer uses data streaming and not TERMCTRL statements in operation. (See "Request Special Terminal Function (4975-01A)" on page LR-334 for information about data streaming on the 4975-01A ASCII printer.)

Syntax Examples

1) Print the contents of the buffer.

```
WRITEPTR  TERMCTRL  DISPLAY
```

2) Set printer to print eight lines per inch.

```
          TERMCTRL  SET,LPI=8
```

3) Set printer to print six lines per inch.

```
          TERMCTRL  SET,LPI=6
```

TERMCTRL - Request special terminal functions *(continued)*

Coding Example

The following example shows three ways in which you can specify a parameter on one of the SET function print operands. In the TERMCTRL instruction labeled T1, the CHARSET operand is coded with the parameter name of the Italian character set (ITAL). In the TERMCTRL instruction labeled T2, the CHARSET operand is coded with an address which contains the equate value for the Italian character set. The MOVEA instruction at label INDEX moves the equate value contained in TABLE into register #1. The CHARSET operand on the TERMCTRL instruction labeled T3 points to a character set at the address defined by the contents of register #1 plus 2.

```
                    .
          COPY      EQU4975
                    .
T1        TERMCTRL  SET,CHARSET=ITAL       CODING THE PARAMETER NAME
T2        TERMCTRL  SET,CHARSET=ITALIAN    CODING AN ADDRESS
INDEX     MOVEA     #1,TABLE
T3        TERMCTRL  SET,CHARSET=(2,#1)     CODING AN INDEXED VALUE
                    .
                    .
TABLE     DATA      A(+$AUGE)              NOTE THAT $AUGE AND $ITAL
ITALIAN   DATA      A(+$ITAL)              ARE EQUATE VALUES
```

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). The supervisor places the address of the instruction that produced the return code in the second word of the TCB (taskname+2).

Code	Description
301	Invalid TERMCTRL statement. Returned for SET function operands PDEN, PMODE, and CHARSET. No terminal error exit is taken.
302	PRINTEXT message exceeds line width. Terminal error exit is taken.

TERMCTRL - Request special terminal functions *(continued)*

4978 Display

Syntax:

```
label         TERMCTRL  function,opnd1,opnd2,count,TYPE=,ATTN=,
                        DCB=
Required:     function
Defaults:     none
Indexable:    opnd1,opnd2
```

Operand ***Description***

function:

BLANK	Prevents displaying input or output characters on the 4978 screen. The contents of the internal buffer remain unchanged. If you specify BLANK, no other operands are required.
DISPLAY	Causes the system to display the screen contents if previously bianked by the BLANK function, to display any buffered output, and to update the cursor position accordingly.
TONE	Causes the system to sound the audible alarm if it is installed.
BLINK	Sets the cursor to the blinking state.
UNBLINK	Sets the cursor to the nonblinking 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 this function is specified, opnd1, opnd2, count, and TYPE= are required.
GETSTORE	Transfers data from storage in the 4978 to the processor. If this function is specified, operands opnd1, opnd2, count, and TYPE are required.

opnd1 The address in the processor from which or to which the data is to be transferred.

TERMCTRL - Request special terminal functions *(continued)*

opnd2 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 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 changed fields. If this option is specified, function must be GETSTORE.

7, to indicate that the field table transferred contains the addresses of the protected portions of changed fields. If this option is specified, function must be GETSTORE.

TERMCTRL - Request special terminal functions *(continued)*

DCB= The label of an 8-word device control block you define with the DCB statement. The 4978 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4978 hardware and terminal I/O internals when you use this operand.

TERMCTRL - Request special terminal functions *(continued)*

Coding Examples

1) The first TERMCTRL instruction prevents the displaying of characters on the 4978 screen. The second TERMCTRL instruction restores the displaying of characters on the screen.

The third TERMCTRL instruction transfers data from storage in the 4978 to the processor.

```
      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
```

2) The following example shows several uses for the TERMCTRL instruction.

```
         TERMCTRL  TONE             ISSUE TONE TO ALERT OPERATOR
         TERMCTRL  UNLOCK           UNLOCK KEYBOARD
         TERMCTRL  BLINK            SET CURSOR TO BLINK MODE
GETID    READTEXT  TXT1,'@ PLEASE ENTER YOUR ID #,LINE=3
         IF        (TXT1-1,EQ,0),GOTO,GETID
         TERMCTRL  UNBLINK          RESET CURSOR TO UNBLINK
*
GETPASS  PRINTEXT  '@ PLEASE ENTER YOUR PASSWORD'
         TERMCTRL  BLANK            INHIBIT DISPLAY OF PASSWORD
         WAIT      KEY              WAIT FOR ENTER KEY
         READTEXT  TXT2             GET USER'S ENTRY
         CALL      CHKPASS          CALL PASSWORD VERIFY ROUTINE
*
         IF        (PASSCHK,NE,-1),GOTO,ENDIT   IF PASSWORD
*                                   DOES NOT MATCH USER ID, EXIT
         TERMCTRL  SET,ATTN=NO      DISABLE ATTENTION KEY
          .
          .
          .
         TERMCTRL  DISPLAY          CLEAR THE BUFFER
*
ENDIT    PRINTEXT  '@ SESSION IS ENDING'
         PRINTEXT  '@ SYSTEM IS AVAILABLE AT 7 AM MON - FRI'
         TERMCTRL  SET  ATTN=YES              ENABLE THE ATTENTION KEY
         TERMCTRL  LOCK                       LOCK THE KEYBOARD
          .
          .
         SUBROUT   CHKPASS,PASSCHK
          .
         RETURN
          .
TXT1     TEXT      LENGTH=30
TXT2     TEXT      LENGTH=30
```

TERMCTRL - Request special terminal functions *(continued)*

4979 Display

Syntax:

```
label        TERMCTRL  function,ATTN=,DCB=

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

Operand	*Description*	
function:		
	BLANK	Prevents displaying input or output characters on the 4979 screen. The contents of the internal buffer remain unchanged. If you specify BLANK, no other operands are required.
	DISPLAY	Causes the system to display the screen contents if previously blanked by the BLANK function, to display any buffered output, and to update the cursor position 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 with the SET function.

DCB= The label of an 8-word device control block you define with the DCB statement. The 4979 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

TERMCTRL - Request special terminal functions *(continued)*

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4979 hardware and terminal I/O internals when you use this operand.

Coding Example

The first TERMCTRL instruction prevents the displaying of characters on the 4979 screen. The second TERMCTRL instruction restores the displaying of characters on the screen.

```
TERMCTRL  BLANK               BLANK SCREEN
    .
    .                         MODIFY DISPLAY
    .
PRINTEXT  LINE=A,SPACES=B     DEFINE CURSOR POSITION
TERMCTRL  DISPLAY             ENABLE DISPLAY
```

TERMCTRL - Request special terminal functions *(continued)*

4980 Display

Syntax:

```
label         TERMCTRL  function,opnd1,opnd2,count,TYPE=,ATTN=,
                        DCB=
Required:     function
Defaults:     none
Indexable:    opnd1,opnd2
```

Operand ***Description***

function:

BLANK	Prevents displaying input or output characters on the 4980 screen. The contents of the internal buffer remain unchanged. If you specify BLANK, no other operands are required.
DISPLAY	Causes the system to display the screen contents if previously blanked by the BLANK function, to display any buffered output, and to update the cursor position accordingly.
TONE	Causes the system to sound the audible alarm if it is installed.
BLINK	Sets the cursor to the blinking state.
UNBLINK	Sets the cursor to the nonblinking 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 4980. If you specify PUTSTORE, opnd1, opnd2, count, and TYPE are required.
GETSTORE	Transfers data from storage in the 4980 to the processor. If you specify GETSTORE, operands opnd1, opnd2, count, and TYPE are required.

opnd1 The address in the processor from which or to which the data is to be transferred.

TERMCTRL - Request special terminal functions *(continued)*

opnd2 The address in 4980 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 with the SET function.

TYPE= You may want to change the image and/or control stores on a 4980 terminal from an application program. For information on doing so, refer to "$RAMSEC - Replace Terminal Control Block (4980)" on page LR-594

1, to show access to the character image buffer (a 4096-byte table, 8 bytes for each of the EBCDIC codes).

2, to show access to the control store.

4, to show 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 show 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 show that the field table transferred contains only the addresses of changed fields. If this option is specified, function must be GETSTORE.

7, to show that the field table transferred contains the addresses of the protected portions of changed fields. If this option is specified, function must be GETSTORE.

8, to show that transfer of the microcode from the processor to the device is in progress.

9, to show that the last segment of the microcode is being sent from the processor to the device.

10, to show that the last segment of the control store is being sent from the processor to the device.

TERMCTRL - Request special terminal functions *(continued)*

For example, to write the last 1024 bytes of the control store (#2 contains the control store address), code the following:

```
TERMCTRL  PUTSTORE,BUFFER,(0,#2),1024,TYPE=10
```

DCB= The label of an 8-word device control block you define with the DCB statement. The 4980 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4980 hardware and terminal I/O internals when you use this operand.

TERMCTRL - Request special terminal functions *(continued)*

5219 Printer

Syntax:

```
label          TERMCTRL  function,STREAM=,LPI= or print operand,
                         DCB=
Required:      function
Defaults:      STREAM=NO
Indexable:     CHARSET,PDEN
```

Operand	*Description*

function:

SET — Sets the number of lines per inch when coded with the LPI operand. If you do not specify the LPI operand, you must code the SET function along with one of the three print operands that allow you to set and control the special print functions available with the 5219 printer. (See "SET Function Operands" on page LR-474 for a description of each of the print operands.)

Note: You must code the SET function along with either the LPI operand or one of the print operands.

DISPLAY — Causes the system to write any buffered output to the printer. No operands are valid with this function.

STREAM= YES, to show that you have already coded the escape sequences the printer needs to do an output operation in the buffer area. For the required escape sequences, refer to the *IBM 5219 Printer Models D01 and D02 Programmer's Reference Guide*, GA23-1025.

NO (the default), to show that the 5219 is in a mode that emulates the 4975 printer.

LPI= The number of lines per inch (either 6 or 8) the printer is to print. Use this operand with the SET function only.

DCB= The label of an 8-word device control block you define with the DCB statement. The printer support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

TERMCTRL - Request special terminal functions *(continued)*

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the printer hardware and terminal I/O internals to use this operand.

SET Function Operands

The three SET function operands allow you to:

- Select the density of printer characters on a line (PDEN).
- Select a language character set (CHARSET).
- Restore the default values for the printer (RESTORE).

You can code only one print operand on each TERMCTRL statement. When specifying parameters on the PDEN and CHARSET operands, you can code the parameter name, an indexed value, or the label of a data area that contains the parameter name. A label must not have the same name as the allowable parameters.

To simplify the coding of labels and indexed values, the system provides an equate table, EQU4975. The parameter equate is the parameter name preceded by a "$" sign. For example, the parameter equate for the Italian character set, ITAL, is $ITAL. Before coding labels or indexed values with the TERMCTRL statement, you must copy the equate module (EQU4975) into your application program with a COPY statement.

Note: To change the print density and character set on a 5219, you must physically change the print wheel. When the PDEN, CHARSET, or RESTORE operands are coded on the TERMCTRL instruction, they cause the 5219 printer to stop printing and signal the operator. At that time, the operator can change the print wheel. The operator must then press the start button to resume printing. Refer to the *IBM Series/1 5219 Printer Models D01 and D02 Setup Procedures/Operator Guide,* GA23-1019, for information on how to change the print wheel.

Operand **Description**

PDEN= Specifies the density of printed characters on each line. You can select "normal" or expanded character density.

Note: All printed characters are of equal width.

NORM — Print in "normal" or typewriter-like characters. You can print up to 198 characters on a line (15 characters per inch).

TERMCTRL - Request special terminal functions *(continued)*

EXPD — Print in expanded characters. You can print up to 132 characters on a line (10 characters per inch).

When you code the PDEN operand, be sure the line length of your TEXT or BUFFER statement does not exceed the maximum line length for the density you choose.

CHARSET= Specifies the language character set the printer uses. The CHARSET operand changes the default character set you specified during system generation. (Refer to the *Installation and System Generation Guide* for the 5219 TERMINAL statement.)

The character set coded with the CHARSET operand becomes the new default for the printer. You can change the default character set with another TERMCTRL statement or with the $TERMUT1 utility. (See the *Operator Commands and Utilities Reference* for details on how to use the $TERMUT1 utility.)

The following character sets are available on the printer:

AUGE Austrian and German
BELG Belgian
BRZL Brazilian
DNNR Danish and Norwegian
FRAN French
FRCA French Canadian
INTL International (multinational)
ITAL Italian
JAEN Japanese and English
KANA Japanese Katakana
PORT Portugese
SPAN Spanish (Spain)
SPNS Spanish (other)
SWFI Swedish and Finnish
UKIN English (United Kingdom)
USCA English (United States and Canada).

RESTORE The PDEN, CHARSET, and LPI operands all cause a permanent change to the defaults established for the printer. The RESTORE operand allows you to restore the default values to the values set with the last CT command of the $TERMUT1 utility or, if the CT command has not been used, to the values specified at system generation time.

When you change printer functions with a TERMCTRL statement, code the RESTORE option on another TERMCTRL statement to restore the original default values.

TERMCTRL - Request special terminal functions *(continued)*

Syntax Examples

1) Print the contents of the buffer.

```
WRITEPTR  TERMCTRL  DISPLAY
```

2) Set printer to print eight lines per inch.

```
TERMCTRL  SET,LPI=8
```

3) Set printer to print six lines per inch.

```
TERMCTRL  SET,LPI=6
```

Coding Example

The following example shows how you can specify the escape sequences for a 5219 printer and turn on data streaming. In the example, the labels M1 through M7 supply the requested printer commands into the buffer. Label M8 is the test message. The forms feed command at label FF is moved into the buffer by the instruction at label M1. This command ejects the printer page. The instruction at label M9 contains the number of words being placed in the buffer. The STREAM operand on the TERMCTRL instruction at label M10 is coded STREAM=YES to show that you have supplied the required escape sequences. If STREAM=NO were coded, the system would supply the default escape sequences. The instructions at labels M11 through M14 reset the printer and turn off data streaming.

Note: The labels M1 through M14 are shown for explanation purposes only and should not be coded in an actual program.

```
        .
        .
        MOVEA     #1,BUFF                      GET BUFFER ADDRESS
M1      MOVE      (0,#1),FF,(1,BYTE)           FORMS FEED
M2      MOVE      (1,#1),SICWP,(5,BYTES)       SET INITIAL CONDITION
                                                    FOR WORD PROCESSING
M3      MOVE      (6,#1),SHF,(4,BYTES)         SET HORIZONTAL FORMAT
M4      MOVE      (10,#1),SVF,(4,BYTES)        SET VERTICAL FORMAT
M5      MOVE      (14,#1),SCD,(6,BYTES)        SET CHARACTER DENSITY
M6      MOVE      (20,#1),SLD,(4,BYTES)        SET LINE DENSITH
M7      MOVE      (24,#1),PPM,(11,BYTES)       PAGE PRESENTATION
M8      MOVE      (35,#1),TESTMSG,(14,BYTES)   MOVE MESSAGE INTO BUFFER
M9      MOVE      BUFFINDX,49                  SET NO. OF BYTES TO PRINT
        ENQT      P5219                        ENQT ON 5219
M10     TERMCTRL  SET,STREAM=YES               TURN ON DATA STREAMING
        PRINTEXT  BUFF                         PRINT
        .
        .
        .
```

TERMCTRL - Request special terminal functions *(continued)*

```
         .
         .
         .
M11      MOVE      (0,#1),FF,(1,BYTE)          FORMS FEED
M12      MOVE      (1,#1,SICDP,(5,BYTES)       RE-SET INITIAL CONDITION
                                                 TO DATA PROCESSING
M13      MOVE      BUFFINDX.6                  SET NO. OF BYTES TO PRINT
         PRINTEXT BUFF                         PRINT
M14      TERMCTRL SET,STREAM=NO                TURN OFF DATA STREAMING
         .
         .
         .
*
FF       DATA    X'0C'             FORMS FEED
SICWP    DATA    X'2BD20345'       INITIAL CONDITION FOR WORD PROCESSING
         DATA    X'01'
SHF      DATA    X'2BC10284'       HORIZONTAL FORMAT OF 132 COLS PER LINE
SVF      DATA    X'2BC2023C'       VERTICAL FORMAT OF 60 LINES PER PAGE
SCD      DATA    X'2BD20429'       CHARACTER DENSITY OF 10 PER INCH
         DATA    X'000A'
SLD      DATA    X'2BC6020C'       LINE DENSITY OF 6 LINES PER INCH
PPM      DATA    X'2BD20948'       PAGE PRESENTATION MEDIA:
         DATA    X'00000102'
*                       |---------- PAPER
*                         |-------- SOURCE DRAWER 2
         DATA    X'000102'
*                     |------------ DESTINATION DRAWER 1
*                       |---------- STANDARD QUALITY
SICDP    DATA    X'2BD20345'       INITIAL CONDITION FOR DATA PROCESSING
         DATA    X'FF'
         .
         .
P5219    IOCB    P5219,BUFFER=BUFF
BUFF     BUFFER  1024,BYTES
BUFFINDX EQU     BUFF-4
BUFFADDR DATA    A(BUFF)
TESTMSG  DATA    CL14'THIS IS A TEST'
         .
         .
```

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). The supervisor places the address of the instruction that produced the return code in the second word of the TCB (taskname+2).

Code	Description
301	Invalid TERMCTRL statement. Returned for SET function operands PDEN and CHARSET. No terminal error exit is taken.
302	PRINTEXT message exceeds line width. Terminal error exit is taken.

TERMCTRL - Request special terminal functions *(continued)*

5224 or 5225 printer

Syntax:

```
 label          TERMCTRL  function,STREAM=,LPI= or print operand,
                          DCB=
Required:      function
Defaults:      STREAM=NO
Indexable:     CHARSET,PDEN
```

Operand ***Description***

function:

SET — Sets the number of lines per inch when coded with the LPI operand. If you do not specify the LPI operand, you must code the SET function along with one of three print operands that allow you to set and control the special print functions available with the 5224 and 5225 printers. (See "SET Function Operands" on page LR-479 for a description of each of the print operands.)

Note: You must code the SET function along with either the LPI operand or one of the print operands.

DISPLAY — Causes the system to write to the printer any buffered output. No operands are valid with this function.

STREAM= YES, to show that you have already coded the escape sequences the printer needs to do an output operation in the text or buffer area. For the required escape sequences, refer to the *IBM Series/1 Printer Attachment 5220 Series Description,* GA34-0242 or the *IBM Series/1 Data streaming Instructions for the 5220 Series Printer Attachment,* GA34-0269.

NO (the default), to show that the system should insert the required escape sequences in the text or buffer area before the printer does an output operation.

LPI= The number of lines per inch (either 6 or 8) the printer is to print. Use this operand only with the SET function.

TERMCTRL - Request special terminal functions *(continued)*

DCB= The label of an 8-word device control block you define with the DCB statement. The printer support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the printer hardware and terminal I/O internals when you use this operand.

SET Function Operands

The three SET function operands allow you to:

- Select the density of printed characters on a line (PDEN).
- Select a language character set (CHARSET).
- Restore the default values for the printer (RESTORE).

You can code only one print operand on each TERMCTRL statement. When specifying parameters on the PDEN and CHARSET operands, you can code the parameter name, an indexed value, or the label of a data area that contains the parameter name. A label must not have the same name as the allowable parameters.

To simplify the coding of labels and indexed values, the system provides an equate table, EQU4975. The parameter equate is the parameter name preceded by a "$" sign. For example, the parameter equate for the Italian character set, ITAL, is $ITAL. Before coding labels or indexed values with the TERMCTRL statement, you must copy the equate module (EQU4975) into your application program with a COPY statement.

Operand **Description**

PDEN= Specifies the density of printed characters on each line. You can select "normal" or expanded character density.

Note: All print characters are of equal width.

NORM — Print in "normal" or typewriter-like characters. You can print up to 198 characters on a line (15 characters per inch).

EXPD — Print in expanded characters. You can print up to 132 characters on a line (10 character per inch).

TERMCTRL - Request special terminal functions *(continued)*

When you code the PDEN= operand, be sure the line length of your TEXT or BUFFER statement does not exceed the maximum line length for the density you choose.

CHARSET= Specifies the language character set the printer uses. The CHARSET operand changes the default character set you specified during system generation. (Refer to the TERMINAL statement for the 5224 and 5225 printers in the &isg).

The character set coded with the CHARSET operand becomes the new default for the printer. You can change the default character set with another TERMCTRL statement or with the $TERMUT1 utility. (See the *Operator Commands and Utilities Reference* for details on how to use the $TERMUT1 utility.)

The following character sets are available on the printer:

AUGE	Austrian and German
BELG	Belgian
BRZL	Brazilian
DNNR	Danish and Norwegian
FRAN	French
FRCA	French Canadian
INTL	International (multinational)
ITAL	Italian
JAEN	Japanese and English
PORT	Portugese
SPAN	Spanish (Spain)
SPNS	Spanish (other)
SWFI	Swedish and Finnish
UKIN	English (United Kingdom)
USCA	English (United States and Canada).

RESTORE The PDEN, CHARSET, and LPI operands all cause a permanent change to the defaults established for the printer. The RESTORE operand allows you to restore the default values to the values set with the last CT command of the $TERMUT1 utility. If the CT command has not been used, it enables restoration to the values specified at system generation time.

When you change printer functions with a TERMCTRL statement, code the RESTORE option on another TERMCTRL statement to restore the original default values before your program ends.

TERMCTRL - Request special terminal functions *(continued)*

Syntax Examples

1) Print the contents of the buffer.

```
WRITEPTR  TERMCTRL  DISPLAY
```

2) Set printer to print eight lines per inch.

```
          TERMCTRL  SET,LPI=8
```

3) Set printer to print six lines per inch.

```
          TERMCTRL  SET,LPI=6
```

Coding Example

The following example shows three ways in which you can specify a parameter on one of the SET function print operands. In the TERMCTRL instruction labeled T1, the CHARSET operand is coded with the parameter name of the Italian character set (ITAL). In the TERMCTRL instruction labeled T2, the CHARSET operand is coded with the label that points to the equate value for the Italian character set. The MOVEA instruction at label INDEX moves the equate value contained in TABLE into register #1. The CHARSET operand on the TERMCTRL instruction labeled T3 points to a character set at the address defined by the contents of register #1 plus 2.

```
             .
          COPY      EQU4975
             .
T1        TERMCTRL  SET,CHARSET=ITAL       CODING THE PARAMETER NAME
T2        TERMCTRL  SET,CHARSET=ITALIAN    CODING AN ADDRESS
INDEX     MOVEA     #1,TABLE
T3        TERMCTRL  SET,CHARSET=(2,#1)     CODING AN INDEXED VALUE
             .
             .
TABLE     DATA      A(+$AUGE)              NOTE THAT $AUGE AND $ITAL
ITALIAN   DATA      A(+$ITAL)              ARE EQUATE VALUES
```

TERMCTRL - Request special terminal functions *(continued)*

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). The supervisor places the address of the instruction that produced the return code in the second word of the TCB (taskname+2).

Code	Description
301	Invalid TERMCTRL statement. Returned for SET function operands PDEN and CHARSET. No terminal error exit is taken.
302	PRINTEXT message exceeds line width. Terminal error exit is taken.

TERMCTRL - Request special terminal functions *(continued)*

ACCA Attached Devices

When your program issues a TERMCTRL instruction to a device attached to an ACCA card, the functions available to your program depend on whether the device uses a modem. If the device uses a modem, you can code all the functions and the ATTN operand.

If a 3101 in block mode is attached to the ACCA card, additional 3101 TERMCTRL functions are available. For a description of those functions see "3101 Display Terminal (Block Mode)" on page LR-450.

Syntax:

```
label          TERMCTRL  function,ATTN=

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

Operand	*Description*	
function:		
	SET	Enables the attention function for the device (when ATTN=YES) or disables the attention function for the device (when ATTN=NO).
	RING	Waits until the modem presents the Ring Indicator (RI) to the Series/1. It provides no timeout.
	RINGT	Waits until the modem presents the Ring Indicator (RI) to the Series/1. If no Ring Indicator (RI) occurs after 60 seconds, this instruction ends and returns an error condition. That information returns to your application program in the first word of the task control block (TCB).
	ENABLE	Activates Data Terminal Ready (DTR) if not jumpered on and waits for the modem to return Data Set Ready (DSR). No timeout is provided.
	ENABLET	Activates Data Terminal Ready (DTR) if not jumpered on and waits for the modem to return Data Set Ready (DSR). If Data Set Ready (DSR) is not returned within 15 seconds, this instruction terminates and returns an error condition. That information returns to your application program in the first word of the task control block (TCB).

TERMCTRL - Request special terminal functions *(continued)*

ENABLEA — Activates Data Terminal Ready (DTR) if not jumpered on and waits for the modem to return Data Set Ready (DSR). When Data Set Ready (DSR) is returned, an answer tone activates 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.

ATTN= NO, to disable the attention and PF key functions.

YES, to enable the attention and PF key functions.

This operand must be used with the SET function.

Coding Example

The TERMCTRL instruction at label T1 waits until the Series/1 receives the Ring Indicator from the modem. At label T2, the TERMCTRL instruction waits for the Data Set Ready indicator. The TERMCTRL instruction at label T3 disconnects the modem.

```
         ENQT      ACCATERM                  ENQUEUE TARGET TERMINAL
         IF        (LINETYPE,EQ,+SWITCHED)  IF SWITCHED
             IF    (DIALTYPE,EQ,+ANSWER)    IF CPU TO ANSWER
T1                 TERMCTRL RING             WAIT FOR RING INTERRUPT
             ENDIF
T2         TERMCTRL ENABLET                  THEN WAIT FOR DATA SET
*                                            READY
         ENDIF
           .
           .
           .
         IF        (LINETYPE,EQ,+SWITCHED)  IF SWITCHED LINE
T3         TERMCTRL  DISABLE                 DISABLE LINE
         ENDIF
         DEQT                                RELEASE THE TERMINAL
         PROGSTOP
DIALTYPE DATA   F'-1'
ANSWER   EQU    0
LINETYPE DATA   F'0'
SWITCHED EQU   -1
ACCATERM IOCB   $SYSLOGA
```

TERMCTRL - Request special terminal functions *(continued)*

General Purpose Interface Bus

The Event Driven Executive provides support for the General Purpose Interface Bus (GPIB) Adapter, RPQ D02118. This support allows an application program to control and access a set of interconnected devices attached to the adapter by a single cable or "bus." These devices could include printers, plotters, graphics display units, and programmable laboratory equipment.

The I/O operations directed to the attached devices and the GPIB bus control are the responsibilities of the application program. The application must, for example, do device selection and polling, and begin all data transfer operations.

For additional details on the GPIB, see the *Communications Guide.*

Syntax:

```
label        TERMCTRL  function,command,options,data

Required:    command
Defaults:    none
Indexable:   data
```

Operand ***Description***

function:

DISPLAY Causes the system to write to the adapter any buffered output. No other operands should be coded with DISPLAY.

GPIB Indicates a GPIB function. The operation is determined by other operands coded on the TERMCTRL instruction.

command:

CON The Configure bus command is used to assign talker/listener roles to devices and can be used to transfer up to 100 bytes of configuration information from programming information. The data delimiter is a double quote and comma (",) and can be used to separate segments of configuration or programming information. The combination double quote and semicolon (":) characters will end the data transfer.

DCL The Device Clear command causes the system to initialize all devices. The initialized state is device dependent.

TERMCTRL - Request special terminal functions *(continued)*

GET	The Group Execute Trigger command causes the specified listener devices to have their predefined basic operation initiated (device dependent).
GTL	The Go To Local command causes the specified listener devices to respond to both the interface message and panel controls.
IFC	The Interface Clear command causes the bus to enter an inactive state. The timer override option cannot be specified with this command.
LLO	The Local Lock Out command causes the specified listener devices to respond to interface control messages but not device panel controls.
MON	The Monitor command allows the transfer of data between devices on the bus. One device must have been previously addressed as a talker and at least one as a listener by a configure operation.
PPD	The Parallel Poll Disable command selectively disables the specified listener devices and prevents them from participating in a parallel poll sequence.
PPE	The Parallel Poll Enable command places the specified listener devices in a response mode.
PPU	The Parallel Poll Unconfigure forces into a parallel poll idle state all devices which are currently able to respond to a parallel poll.
READ	The Read command allows the transfer of data into storage from a device on the bus. The device must previously have been assigned as a talker. Any listener devices will receive the data, also.
REN	The Remote Enable command allows specified listener devices to respond to further operations.
RPPL	The Parallel Poll Results command reads the result of the latest parallel poll into storage. The address specified in the data operand contains the results and is returned as one byte.
RSB	The read adapter Residual Status Block operation retrieves an adapter status block after an operation which requested suppress exception (SE). The status information is returned in the location specified by the data operand of the TERMCTRL instruction.
RSET	The Reset Adapter command resets the GPIB adapter and clears any pending interrupts.

TERMCTRL - Request special terminal functions *(continued)*

SDC	The Selected Device Clear command causes the system to reset the specified listener devices.
SPD	The Serial Poll Disable command disables the serial poll status reporting ability of the devices previously enabled.
SPE	The Serial Poll Enable command initializes the specified talker devices to present status in response to a parallel poll.
SPL	Serial Poll Status reads the results of the latest serial poll into storage.
STAT	Read Adapter Cycle Steal Status returns the GPIB adapter cycle steal status resulting from a previous operation. The status information is returned in the storage location indicated by the data operand of this command.
WPPL	The Write Parallel Poll command does a parallel poll of the devices that were previously enabled by a PPE command.
WRIT	A Write Data operation places device programming information or data on the bus for those devices specified as listeners.

options: When using more than one option, separate them with commas and enclose them all in parentheses.

EOI	The end-or-identity terminator is a signal used by a talker to indicate the last byte of a block of data. The adapter ends a read operation with fewer than the specified number of characters if a talker signals an end-or-identity condition. The adapter can establish an EOI condition by requesting the EOI option. EOI is valid for the following commands: CON, MON, READ, and WRIT. You may not specify EOI together with the end-of-string (EOS) option.
EOS	Encountering an end-of-string terminator ends a read operation immediately. EOS is valid only for the MON and READ commands, but it cannot be coded in the same instruction with the EOI option.
SE	The Suppress Exception prevents the reporting of exception conditions because of incorrect length records (ILR). An ILR exception occurs when a GPIB read is ended with fewer than the specified number of characters read. The contents of the residual status block (RSB) is meaningful only for this condition. SE is valid only for the commands MON and READ.

TERMCTRL - Request special terminal functions *(continued)*

TO — The Timer Override option causes the adapter to wait for an operation to complete. All GPIB commands can specify TO except for RSET, RSB, STAT, IFC, WPPL, RPPL, and SPL.

data — Use this operand to specify additional information for the commands STAT, RSB, or RPPL, or for the option EOS.

Use it to specify the label of an address where a program will store status data when you code it with commands STAT, RSB, or RPPL.

Specify either the EOS character or the address of a word which contains, in bits 8 - 15, the EOS character when you use it with the EOS option.

TERMCTRL - Request special terminal functions *(continued)*

Series/1-to-Series/1

The Event Driven Executive provides support for the Series/1-to-Series/1 Attachment, RPQ D02241 and RPQ D02242. This attachment allows an application to communicate with two or more Series/1 processors over a communications link.

Either Series/1 processor can begin a data transfer operation. To complete data transfer operations, issue a read (READTEXT), write (PRINTEXT), or control (TERMCTRL) instruction through an application program. Call the issuing processor the "initiating" processor. Call the processor that must respond with the opposite instruction the "responding" processor.

For TERMCTRL operations, the required state of the "other" processor (initiating or responding) depends on the particular type of TERMCTRL operation you want to perform.

Syntax:

```
label        TERMCTRL  function,opnd1,opnd2,count,WAIT=

Required:    function
Defaults:    WAIT=NO
Indexable:   opnd1,opnd2
```

Operand ***Description***

function:

ABORT Causes a Write ABORT operation. The responding processor will cause the operation on the beginning processor to end the last operation. A return code of 1010 is returned in the task code word. If the operation is attempted but no request is pending from the initiating processor, an error code is returned.

Both the initiating and responding processors must have active Series1-to-Series/1 application programs for this request to be meaningful. The ABORT function is only valid for the responding processor.

IPL Causes the initiating processor to send an IPL request to the responding processor. The processor initiating the IPL transfers from the address opnd1 indicates, the number of bytes its count operand specifies. Opnd2 indicates the the address key from which the storage load will be sent.

The responding processor receives a system reset from the attachment then enters load mode and receives the storage load.

TERMCTRL - Request special terminal functions *(continued)*

RESET — Causes a device reset to the attachment specified by the most recent ENQT instruction. This will clear any pending interrupt or busy condition.

RESET can be issued anytime, by either processor, regardless of the state of the other processor.

STATUS — Obtains status information from the responding processor. Opnd1 specifies the address of a two-word block of storage which will receive the header data. The header data represents requests the initiating processor issues. If you code opnd2, it is the target address of the diagnostic jumper word plus the 11 cycle steal status words. Read cycle steal status words only following an error. Normally, the contents will be zero.

opnd1 — Use this operand with the IPL and STATUS functions. When you use it with IPL, it specifies the address from which you wish to send the storage load to the responding processor.

When you use opnd1 with the STATUS function, it specifies an address where the two-word header is to be stored.

You can use the contents of the 2-word header to determine the attached processor operations as follows:

Word 1	bits 0 - 1	= 0	
	bit 2	= 0	The responding processor has issued a READTEXT
		= 1	The responding processor has issued a PRINTEXT
	bits 4 - 7	=	Checksum value
	bits 8 - 15	= 0	

Word 2 — Specifies the number of bytes to be transferred.

TERMCTRL - Request special terminal functions *(continued)*

opnd2 Use this operand with the IPL and STATUS functions. When ypu use it with IPL, it specifies the address key for the storage load. Code an integer specifying the address key (the partition number minus 1).

When you use this operand with the STATUS function, it specifies two addresses. One is the address in which to place the 1-word jumper status. The other is the 11-word cycle steal status information.

The status words can be used to determine the status of the attachments as follows:

Word 0	jumper word		
	bits 0 - 7	= 00000000	= RPQ D02242
		00000001	= RPQ D02241
		00000010	= RPQ D02241
		00000011	= invalid
	bit 8	= RPQ D02241 is active	
	bit 9	= RPQ D02242 is active	

Words 1-12 contain the attachment cycle steal status.
These words will be zero unless an error has occurred on the device.

Note: *IBM Series/1-to-Series/1 Attachment RPQs D02241 & D02242 Custom Feature*, GA34-1561 provides further descriptions of the bit settings and the contents of words 1 - 12.

count The count operand is used with the IPL function to specify the number of bytes to be sent to the processor receiving the IPL.

WAIT This operand, when coded WAIT=YES, prevents control from being returned to the initiating processor until the responding processor issues a successful READTEXT or PRINTEXT operation. Note that neither a TERMCTRL ABORT nor TERMCTRL RESET can override this operand when it is coded WAIT=YES. The default for this operand is WAIT=NO.

TERMCTRL - Request special terminal functions *(continued)*

Teletypewriter Attached Devices

This can be a teletypewriter-equivalent device such as a 3101 operated in character mode or an ASR 33/35 connected to a teletypewriter adapter.

Syntax:

```
 label          TERMCTRL  function,ATTN=

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

Operand   ***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 teletypewriter.

ATTN=   NO, to disable the attention function.

YES, to enable the attention function.

This operand must be used with the SET function.

Syntax Examples

1) Display the contents of the buffer.

```
TERMCTRL  DISPLAY          DISPLAY THE BUFFER
```

2) Disable the attention key function.

```
TERMCTRL  SET,ATTN=NO
```

3) Enable the attention key function.

```
TERMCTRL  SET,ATTN=YES
```

TERMCTRL - Request special terminal functions *(continued)*

Virtual Terminal

Virtual terminal support uses the PRINTEXT and READTEXT instructions to communicate between programs. It requires two TERMINAL configuration statements and the supervisor module IOSVIRT. Virtual terminal support provides synchronization logic. For details on virtual terminal other than TERMCTRL operands, refer to the *Communications Guide*.

Syntax:

```
label        TERMCTRL  function,code,ATTN=

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

Operand	*Description*	
function:		
	DISPLAY	Causes any buffered output to be transmitted across the virtual channel.
	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 nonzero value, "x", the attention list code $PFx is automatically generated and the ">" response does not occur. 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.	

TERMCTRL - Request special terminal functions *(continued)*

ATTN= NO, disables attention function acknowledgement by the system.

YES, enables attention function acknowledgement by the system.

A systems ability to send attention interrupts is not affected in either case. Each setting of this operand controls terminal operations until reset.

This operand must be used with the SET function.

Coding Examples

1) The following example may be used for program communication using virtual terminal support when attention list processing is implemented with the PF key evaluation.

The TERMCTRL instruction at label T1 disables the attention key for the virtual terminal device. At label T2, the TERMCTRL instruction presents a program function key interrupt.

```
          ENQT      B                            GET VIRTUAL CHANNEL B
          LOAD      PGM4,LOGMSG=NO               LOAD COMMUNICATING PGM
          ENQT      A                            GET VIRTUAL CHANNEL A
T1        TERMCTRL  SET,ATTN=NO                  DISABLE ATTENTION KEY
          READTEXT  LINE,MODE=LINE               GET OUTPUT FROM PGM4
          TCBGET    RETURNCD,$TCBCO              GET RETURN CODE
          DEQT      A                            RELEASE CHANNEL A
          IF        (RETURNCD,EQ,5),GOTO,ENDIT
*                                                IF PGM4 ENDED, STOP
            IF      (LINE,EQ,ENTRCMD,(13,BYTE))  IF PGM4
*                                                REQUESTS INPUT COMMAND
T2            TERMCTRL PF,4                      SEND PF4 (SEARCH VOLUME)
            ENDIF
             .
             .
             .
            PROGSTOP
ENTRCMD     DATA    C'ENTER COMMAND'
```

TERMCTRL - Request special terminal functions *(continued)*

2) The following example may be used for program communication using virtual terminal support when attention list processing is implemented with the PF key evaluation.

Consider the following main program example for ease of coding. In it, two subroutines manage the virtual terminal on the companion side of the channel which will be referred to as the B side.

```
TASK     PROGRAM START
START    EQU     *
         .
         ENQT    VIRTB                     ENQUEUE ON B SIDE OF CHANNEL
         LOAD    (PGM4,EDX003),LOGMSG=NO  LOAD PROGRAM
         ENQT    VIRTA                     ENQUEUE ON A SIDE OF CHANNEL
         MOVE    MESSAGE,TST,(4,BYTES)     INITIALIZE ATTENTION LIST CMD.
         *
         CALL    SENDCMD                   GO SEND COMMAND TO B SIDE
          .                              CMD FOR THE B SIDE
         PROGSTOP                         OF CHANNEL.
VIRTA    IOCB    CDRVTA
VIRTB    IOCB    CDRVTB
MESSAGE  TEXT    LENGTH=8
TST      TEXT    C'$A 3'
BUFFER   TEXT    LENGTH=80
CARET    DATA    C'>'
```

3) The following subroutine handles transmission of attention list processing commands destined for the B side of the channel.

```
     SUBROUT   SENDCMD
            TERMCTRL     PF,0              SEND ATTENTION TO B SIDE
            READTEXT BUFFER,MODE=LINE      READ RESPONSE FROM B SIDE
  IF
            (TASK,EQ,5)                    IF THIS IS AN END OF
                                           ATTENTION OR PROGSTOP
  CALL        PLACE                        GO CORRECT PARTITION
  ENDIF
  IF        (TASK,EQ,-1),OR,(TASK,EQ,-2)  IF RETURN CODE GOOD
    IF         (BUFFER,EQ,CARROT)          AND GOT THE '>' SIGN
  PRINTEXT        MESSAGE                  SEND THE ATTENTION LIST
                                           COMMAND TO THE B SIDE
  DO                UNTIL,(TASK,EQ,5)      CHECK FOR THE B SIDE
      READTEXT         BUFFER, MODE=LINE   ATTENTION LIST PROCESSING
          .
        ENDDO
      ELSE
        ERROR PROCESS
          .
      ENDIF
    ELSE
      ERROR PROCESS
          .
    ENDIF
    RETURN                                 RETURN TO CALLING PROGRAM.
```

TERMCTRL - Request special terminal functions *(continued)*

4) The following subroutine handles recovery of the address space for the keyboard task on the B side of the channel. This occurs at a progstop as a result of secondary and tertiary program loads.

```
SUBROUT   PLACE
    IF         (BUFFER,EQ,CARROT) IF RESPONSE IS '>'
       PRINTEXT    "$CP X"               RECOVER THE PARTITION
       TERMCTRL    DISPLAY               SEND COMMAND
       DO          UNTIL,(TASK,EQ,5)     CHECK FOR THE END OF
       READTEXT    BUFFER,MODE=LINE      ATTENTION LIST PROCESSING
       ENDDO
       TERMCTRL     PF,0                 SEND ATTENTION TO THE B SIDE
       READTEXT    BUFFER,MODE=LINE      READ RESPONSE
     ENDIF
     RETURN                              RETURN TO THE CALLER
     ENDPROG
     END
```

TEXT - Define a text message or text buffer

The TEXT statement defines a message or a storage area for character data. You can store character data in either EBCDIC or ASCII code.

You can use the PRINTEXT instruction to print or display a message on a terminal. The READTEXT instruction can be used to read a character string from a terminal into the storage area defined by the TEXT statement.

READTEXT and GETEDIT instructions described in this manual may be used to modify the TEXT statement. PRINTEXT and PUTEDIT instructions, also described in this manual, use the TEXT statement to determine the number of values to print.

In storage, the first word of each TEXT statement contains a length byte and a count byte. The length byte (byte 0) contains the size of the storage area in bytes. The count byte (byte 1) shows the actual number of characters in the storage area.

Figure 10 on page LR-499 shows the structure of the TEXT statement.

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
```

Operand	*Description*
label	The label of the first byte of text. The GETEDIT, PUTEDIT, READTEXT, and PRINTEXT instructions refer to this label.
'message'	Any character string defined between apostrophes. The count field will equal the actual number of characters between apostrophes.

If you do not code this operand, you must code LENGTH, and the storage area is filled with EBCDIC blanks. You should not code this operand if you use the storage area initially for input.

If the LENGTH operand is not coded and the count value is even, then LENGTH=count. However, if the count value is odd, then LENGTH=count+1.

Use two apostrophes to represent each printable apostrophe.

TEXT - Define a text message or text buffer *(continued)*

The symbol "@" causes a carriage return or line feed to occur on roll screen terminals.

LENGTH= The size (in bytes) of the storage area. The maximum value you can code is 254. If you do not code this operand, you must code the 'message' operand, and LENGTH equals the number of characters between the apostrophes.

The system truncates messages that exceed the length of the storage area. If the message does not fill the storage area, the system pads the area to the right of message with EBCDIC blanks.

Note: With $S1ASM, TEXT has a maximum length of 98 and a default length of 64.

If you do not code the 'message' operand, the system fills the storage area with EBCDIC blanks and the count byte is equal to the length byte.

CODE= Defines the data type. Code E for EBCDIC or A for ASCII. E is the default.

Syntax Examples

1) The PRINTEXT instruction displays the phrase "A MESSAGE" on a terminal.

```
         .
         .
         PRINTEXT   MSG1
         .
         .
MSG1     TEXT       'A MESSAGE'
         .
         .
```

2) The PRINTEXT instruction displays the phrase "ABC   " on a terminal. Because the text buffer length is 10 bytes and the message is only 3 bytes long, the system fills the buffer space to the right of the message with blanks. CODE=A sets the character date type to ASCII.

```
         .
         .
         PRINTEXT  MSG2
         .
         .
         PROGSTOP
MSG2     TEXT      'ABC',LENGTH=10,CODE=A
         .
         .
```

TEXT - Define a text message or text buffer *(continued)*

3) The READTEXT instruction waits for a response entered from a terminal. The system will place the response in the TEXT statement labeled MSG#. If the response has fewer than 30 characters, the system pads the storage area to a length of 30 bytes. If the response is more than 30 characters, the system truncates it after reading 30 bytes.

```
                  .
                  .
          READTEXT  MSG#,'ENTER YOUR HOMETOWN'
                  .
                  .
          PROGSTOP
MSG#      TEXT      LENGTH=30
                  .
                  .
```

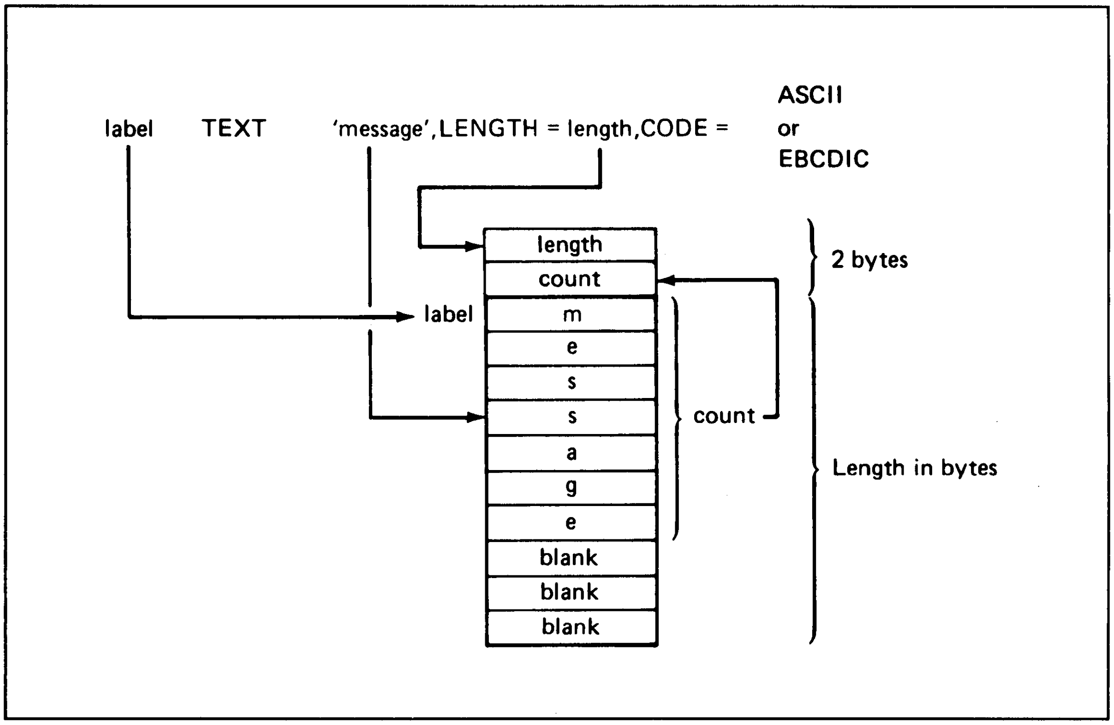

Figure 10. TEXT Statement

TITLE - Place a title on a compiler listing

The TITLE statement places a title at the top of each page of the compiler listing. A program can contain more than one TITLE statement. Each statement generates a new title on the page that follows it. The system repeats this title on each page until it encounters another TITLE statement.

Syntax:

```
blank        TITLE     message

Required:    message
Defaults:    none
```

Operand	*Description*
message	For the macro and host assemblers, you can code an alphameric character string up to 100 characters in length. The string must be enclosed in apostrophes. The $EDXASM compiler will accept an alphameric string of up to 48 characters. The string must be enclosed in apostrophes and must be all on one line.

Coding Example

See the PRINT statement for an example using TITLE.

TP Instruction - Perform Host Communications Facility Operations

The Host Communications Facility instruction (TP) can do the following operations:

- Write to a host data set (TP WRITE)
- Read from a host data set (TP READ)
- Submit a background job to the host system (TP SUBMIT)
- Get the time and date from the host system (TP TIMEDATE)
- Set the occurrence of a Series/1 event so it can be tested by a program running on the host system (TP SET)
- Test for the occurrence of an event set by the host system (TP FETCH)
- Erase the record, on the host system, of an event that occurred on either the Series/1 or the host system (TP RELEASE.)

You do each operation using a different format of the TP instruction. Other TP instruction formats prepare the Series/1 to do an operation (TP OPENIN/TP OPENOUT) or end an operation (TP CLOSE). Each of the TP formats is described in the following section. Refer to the *Communications Guide* for sample programs using the TP instruction formats.

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (CLOSE) - End a transfer operation

TP CLOSE ends a transfer operation. Use this instruction to end an operation begun with TP OPENOUT or with TP OPENIN.

Notes:

1. If an error occurs, the system automatically closes an open data set. The only time you must issue a TP CLOSE is when a data set transfer is being ended and no errors have occurred. This situation would occur, for instance, if only 10 records were being written to or read from a data set capable of containing 20 records.

2. Always test the return code after you issue a TP CLOSE because some errors are only detected at this time (return codes 50 and 51, for example).

3. While you have an open data set, no one else is able to use the facility.

Syntax:

```
label          TP          CLOSE,ERROR=

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

Operand	*Description*
CLOSE	Code as shown.
ERROR=	The label of the first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control passes to the next sequential instruction and you must test for errors.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (FETCH) - Test for a record in the system-status data set

TP FETCH tests for the existence of a specific record in the System-Status Data Set on the host system and, optionally, reads in the associated data record.

Syntax:

```
label        TP        FETCH,stloc,length,ERROR=,P2=,P3=

Required:    FETCH,stloc
Defaults:    length=0
Indexable:   stloc,length
```

Operand	*Description*
FETCH	Code as shown.
stloc	The label of a STATUS instruction. See the STATUS instruction for more details.
length	Specify the length, in bytes, of the data portion of the status record to be received. A count of zero indicates that no data is to be received. The maximum value of this field is 256.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (OPENIN) - Prepare to read data from a host data set

TP OPENIN prepares the Series/1 to read data from a host data set.

Syntax:

```
label         TP          OPENIN,dsnloc,ERROR=,P2=

Required:     OPENIN,dsnloc
Defaults:     none
Indexable:    dsnloc
```

Operand	*Description*
OPENIN	Code as shown.
dsnloc	The label of a TEXT statement that specifies the name of a host data set of standard format. The data set can be a sequential data set or a partitioned data set with member name included.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.
P2=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (OPENOUT) - Prepare to transfer data to a host data set

TP OPENOUT prepares the Series/1 to transfer data to a host data set.

Syntax:

```
label         TP         OPENOUT,dsnloc,ERROR=,P2=

Required:     OPENOUT,dsnloc
Defaults:     none
Indexable:    dsnloc
```

Operand	*Description*
OPENOUT	Code as shown.
dsnloc	The label of a TEXT statement that specifies the name of a host data set of standard format. The data set can be a sequential data set or a partitioned data set with member name included.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.
P2=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (READ) - Read a record from the host

TP READ reads a data record from the host system.

Syntax:

```
label        TP       READ,buffer,count,END=,ERROR=,P2=,P3=

 Required:    READ,buffer
 Defaults:    count=256
 Indexable:   buffer,count
```

Operand	*Description*
READ	Code as shown.
buffer	The label of the data buffer where the record is to be stored. This buffer should be generated with, or should conform to the specifications of, a BUFFER statement specifying TPBSC.
count	The maximum number of bytes to be read. For variable-length records, this count includes the 4-byte Record Descriptor Word (RDW). Refer to the *Communications Guide* for more details on variable-length records.
END=	The first instruction of the routine to be invoked if an "end-of-data-set" condition is detected (return code 300). If you do not specify this operand, the system treats the end of data set condition as an error.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If you do not specify this operand, control is returned to the next sequential instruction and you must test for errors.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (RELEASE) - Delete a record in the system-status data set

TP RELEASE deletes a specific record in the System-Status Data Set on the host system and, optionally, reads the associated data record.

Syntax:

```
label         TP         RELEASE,stloc,length,ERROR=,P2=,P3=

Required:     RELEASE,stloc
Defaults:     length=0
Indexable:    stloc,length
```

Operand	*Description*
RELEASE	Code as shown.
stloc	The label of a STATUS instruction. See the STATUS instruction for more details.
length	Specify the length, in bytes, of the data portion of the status record to be received. A count of zero indicates that no data is to be received. The maximum value of this field is 256.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (SET) - Write a record in the system-status data set

TP SET writes a record in the System-Status Data Set on the host system.

Syntax:

```
label          TP          SET,stloc,length,ERROR=,P2=,P3=

 Required:     SET,stloc
 Defaults:     length=0
 Indexable:    stloc,length
```

Operand	*Description*
SET	Code as shown.
stloc	The label of a STATUS instruction. See the STATUS instruction for more details.
length	Specify the length, in bytes, of the data portion of the status record to be transmitted. A count of zero indicates that no data is to be transmitted. The maximum value of this field is 256.
ERROR=	The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (SUBMIT) - Submit a job to the host

TP SUBMIT submits a job from the Series/1 to the host batch job stream.

Syntax:

```
label        TP        SUBMIT,dsnloc,ERROR=,P2=

Required:    SUBMIT,dsnloc
Defaults:    none
Indexable:   dsnloc
```

Operand	*Description*
SUBMIT	Code as shown.
dsnloc	The label of a TEXT statement that specifies the name of a host data set containing the job (JCL and optional data) to be submitted. You can code either:

- TEXT "dsname" for a sequential data set, or
- TEXT "dsname(membername)" for a partitioned data set.

In systems with a HASP/Host Communications Facility interface, specifying DIRECT for dsnloc allows immediate transmission of data records to the job stream without using an intermediate host data set. To use this facility, code the following:

```
                  .
                  .
         TP   SUBMIT,DIRECT
         TP   WRITE,buffer,80
*
* Code one    TP WRITE,buffer,80   for each job stream record
*
         TP   CLOSE
```

ERROR= The first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control is returned to the next sequential instruction and you must test for errors.

P2= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

TP Instruction - Perform Host Communications Facility Operations *(continued)*

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (TIMEDATE) - Get time and date from the host

TP TIMEDATE obtains the time of day (hours, minutes, and seconds) and the date (month, day, and year) from the host system.

Syntax:

```
label        TP        TIMEDATE,loc,ERROR=,P2=

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

Operand	*Description*
TIMEDATE	Code as shown.
loc	The label of a 6-word data area where time of day and date are stored in the order: hours, minutes, seconds, month, day, and year.
ERROR=	The label of the first instruction of the routine to be invoked if an error condition occurs during this operation. If you do not code this operand, control passes to the next sequential instruction and you must test for errors.
P2=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

Return Codes

All return codes for the TP instruction are listed under TP (WRITE).

TP Instruction - Perform Host Communications Facility Operations *(continued)*

TP (WRITE) - Write a record to the host

TP WRITE sends a data record to the host system.

Syntax:

```
label        TP        WRITE,buffer,count,END=,ERROR=,P2=,P3=

Required:   WRITE,buffer
Defaults:   count=256
Indexable:  buffer,count
```

Operand	*Description*
WRITE	Code as shown.
buffer	The label of the data buffer that contains the record to be transmitted. This buffer should be generated with, or should conform to the specifications of, a BUFFER statement specifying TPBSC.
count	The number of Series/1 bytes to be transferred. For variable-length records, this includes the 4-byte Record Descriptor Word (RDW).
END=	The label of the first instruction of the routine to be invoked if the system detects an "end of data set" (EOD) condition (return code 400). If this operand is not specified, the system treats an EOD as an error.
ERROR=	The label of 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 passes to the next sequential instruction and you must test for errors.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

TP Instruction - Perform Host Communications Facility Operations *(continued)*

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname). Because program execution halts until the operation is complete, your program must test the return code to determine if the operation is successful.

Note: If an error is detected, an open data set is automatically closed for you.

Code	Description	Module
-1	Successful completion	Supervisor
1	Illegal command sequence	Supervisor
2	TP I/O error	Supervisor
3	TP I/O error on host	HCFCOMM
4	Looping bidding for the line	Supervisor
5	Host acknowledgement to request code was neither ACK0, ACK1, WACK, or a NACK	Supervisor
6	Retry count exhausted - last error was a timeout: the host must be down	Supervisor
7	Looping while reading data from the host	Supervisor
8	The host responded with other than an EOT or an ENQ when an EOT was expected	Supervisor
9	Retry count exhausted - last error was a modem interface check	Supervisor
10	Retry count exhausted - last error was not a timeout, modem check, block check, or overrun	Supervisor
11	Retry count exhausted - last error was a transmit overrun	Supervisor
50	I/O error from last I/O in DSWRITE	DSCLOSE
51	I/O error when writing the last buffer	DSCLOSE
100	Length of DSNAME is zero	HCFCOMM
101	Length of DSNAME exceeds 52	HCFCOMM
102	Invalid length specified for I/O	HCFINIT

TP Instruction - Perform Host Communications Facility Operations *(continued)*

Code	Condition	Module
200	Data set not on volume specified for controller	HCFINIT
201	Invalid member name specification	DSOPEN
202	Data set in use by another job	DSOPEN
203	Data set already allocated to this task	DSOPEN
204	Data set is not cataloged	DSOPEN
205	Data set resides on multiple volumes	DSOPEN
206	Data set is not on a direct access device	DSOPEN
207	Volume not mounted (archived)	DSOPEN
208	Device not online	DSOPEN
209	Data set does not exist	DSOPEN
211	Record format is not supported	DSOPEN
212	Invalid logical record length	DSOPEN
213	Invalid block size	DSOPEN
216	Data set organization is partitioned and no member name was specified	DSOPEN
217	Data set organization is sequential and a member name was specified	DSOPEN
218	Error during OS/ OPEN	DSOPEN
219	The specified member was not found	DSOPEN
220	An I/O error occurred during a directory search	DSOPEN
221	Invalid data set organization	DSOPEN
222	Insufficient I/O buffer space available	DSOPEN
300	End of an input data set	DSREAD
301	I/O error during an OS/ READ	DSREAD
302	Input data set is not open	DSREAD
303	A previous error has occurred	DSREAD

TP Instruction - Perform Host Communications Facility Operations *(continued)*

Code	Condition	Module
400	End of an output data set	DSWRITE
401	I/O error during an OS/ WRITE	DSWRITE
402	Output data set is not open	DSWRITE
403	A previous error has occurred	DSWRITE
404	Partitioned data set is full	DSCLOSE
700	Index, key,and status record added	SET
701	Index exists, key and status added	SET
702	Index and key exist, status replaced	SET
703	Error - Index full	SET
704	Error - Data set full	SET
710	I/O Error	SET
800	Index and key exist	FETCH
801	Index does not exist	FETCH
802	Key does not exist	FETCH
810	I/O error	FETCH
900	Index and/or key released	RELEASE
901	Index does not exist	RELEASE
902	Key does not exist	RELEASE
910	I/O error	RELEASE
1xxx	An error occurred in a subordinate module during SUBMIT. 'xxx' is the code returned by that module.	S7SUBMIT

USER - Use assembler code in an EDL program

The USER instruction allows you to use Series/1 assembler code within an EDL program.

Do not use Series/1 Assembler routines to issue input/output instructions to Series/1 standard devices. Use only standard Event Driven Language input/output instructions.

Your Series/1 assembler routine uses a set of hardware registers to do operations. You should save the contents of these registers on entry into the routine. You must restore the register contents before returning control to the EDL program. Details of the conventions that must be followed are described under "Considerations when Coding Assembler Routines."

Syntax:

```
label          USER        name,PARM=(parm1,...,parmn),
                           P=(name1,...,namen)

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

Operand	***Description***
name	The entry point name of your Series/1 assembler routine.
PARM=	A list of parameters that are to be passed to your routine.
P=	A list of names to be attached to the PARM operands.

Considerations when Coding Assembler Routines

On entry to the Series/1 assembler routine, hardware register 1 points to your first parameter. If no parameters are passed to the routine, register 1 points to the address of the next instruction following the USER instruction. Hardware register 2 contains the address of the current task's TCB. Your routine must preserve the contents of register 2 for eventual return to the supervisor. The 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 routine, register 1 must be increased within your routine by double the number of parameters used before returning to the supervisor. If you want to return to an instruction other than the instruction following the USER instruction, you can set register 1 to the address of the desired instruction. In all cases, the assembly language routine must exit by a branch to the label RETURN.

USER - Use assembler code in an EDL program *(continued)*

The USER instruction requires one of the following:

- Allowing the RETURN= operand on the ENDPROG statement in your program to default to RETURN=YES
- $EDXLINK used to include the $$RETURN and the $$SVC object modules.

The autocall feature of $EDXLINK also can be used. Refer to the *Event Driven Executive Language Programming Guide* for additional information on $EDXLINK.

Figure 11 shows the control flow to and from a Series/1 assembler routine.

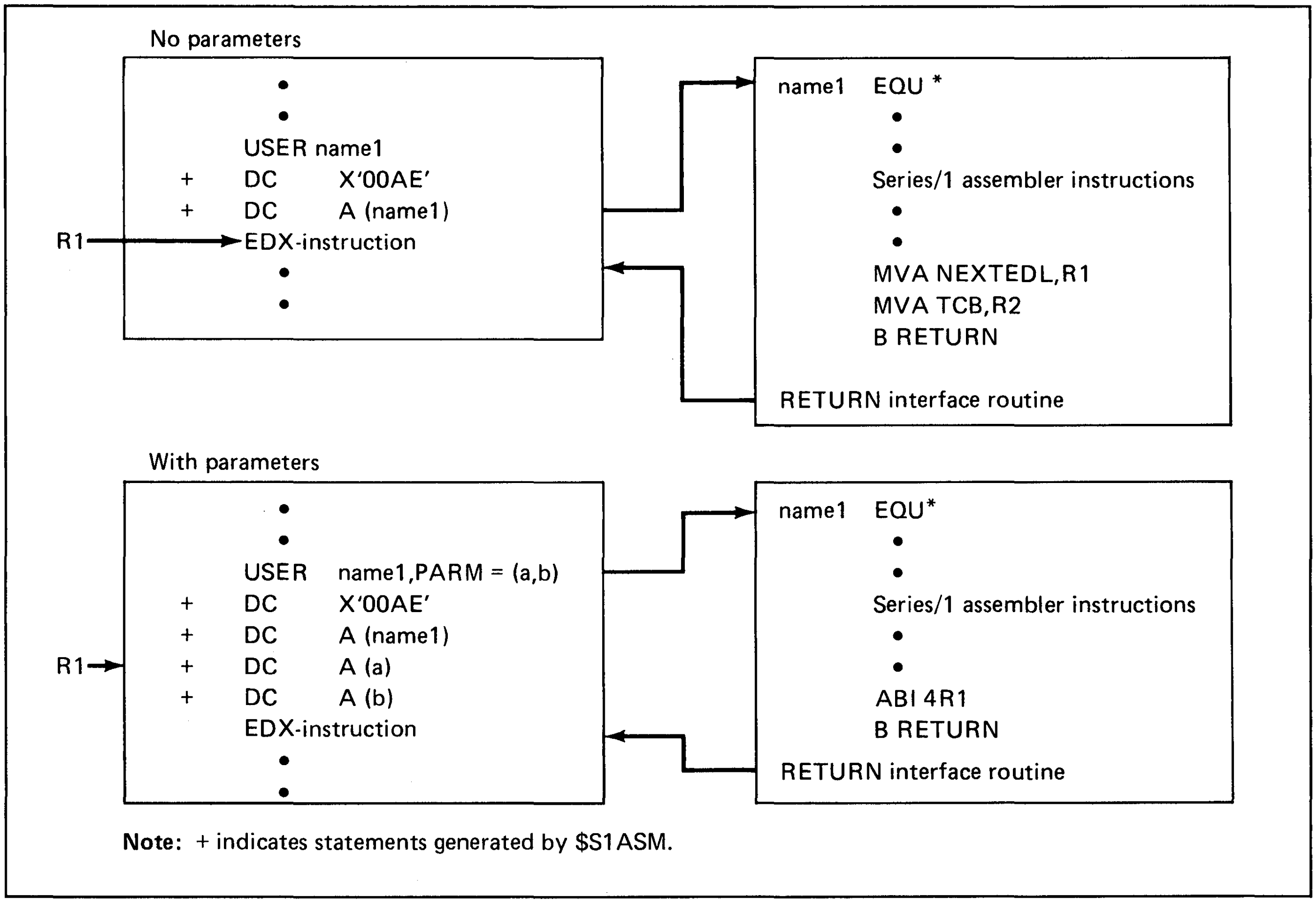

Figure 11. Calling a Series/1 Assembler Routine and Returning

You can pass parameters as constants, which will be stored in the calling list, or pass the symbolic names (addresses) of the parameters. In the latter case, the address of the parameter is contained in the calling list.

USER - Use assembler code in an EDL program *(continued)*

If the parameter is a constant, it may be addressed through hardware 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.

USER - Use assembler code in an EDL program *(continued)*

The second parameter also can be loaded by:

```
         MVW    (R1,2),R3
```

The following instruction shows how to acquire a parameter (in this case, the second) whose address is passed in the calling sequence.

```
         MVW    (R1,2)*,R3
```

Your routine is free to use all the registers if registers 1 and 2 are set properly for return to the supervisor. The last instruction of your routine must branch to RETURN which is an entry point in the interface module $$RETURN. You must link-edit this module to the assembler routine with the $EDXLINK utility.

In the following example, an EDL program passes control to a Series/1 assembler routine with USER *+2. The routine passes control back to the EDL program with BAL RETURN,R1.

```
          .
          .
          .
         MOVE     A,B            STANDARD INSTRUCTION EXAMPLE
         ADD      A,10           ANOTHER INSTRUCTION
*
         USER     *+2            ENTRY TO ASSEMBLER CODE
         MVW      R2,SAVER2      SAVE HARDWARE REGISTER 2 (TCB)
          .
          .                      ASSEMBLER CODE
          .
OK       EQU      *
         MVW      SAVER2,R2      RESTORE HARDWARE REGISTER 2 (TCB)
         BAL      RETURN,R1      SET HARDWARE REGISTER 1 AND RETURN
*
         MOVE     B,A            NOW BACK INTO THE EDL PROGRAM
         SUB      B,10
          .
          .
          .
```

If your EDL program contains assembler code, you must assemble the program using the Series/1 Macro or host assemblers. $EDXASM does not allow mixing Series/1 code with the Event Driven Language instructions. If your assembler routine is in a separate module, you must assemble the routine using one of the macro assemblers and link-edit that module to the EDL program with $EDXLINK.

For information regarding use of the USER command in logging errors refer to "$USRLOG - Log Specific Errors From a Program" on page LR-599.

WAIT - Wait for an event to occur

The WAIT instruction allows your program to wait for an event to occur, such as 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.

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 the WAIT instruction. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example that waits for an event to occur in another partition. For more information on cross-partition services, refer to the *Event Driven Executive Language Programming Guide.*

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 explicitly coded unless one of the system event names previously described is used (PIx, TIMER, DSn, and so on). When the WAIT is satisfied with a POST instruction, the post code is stored in both the ECB and the waiting task's TCB code location.

Syntax:

```
label          WAIT     event,RESET,P1=

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

Operand	***Description***
event	The label of the event for which the system is waiting.

For process interrupt, use PIx, where "x" is a user process interrupt number in the range 1-99.

For intervals set by STIMER, use TIMER as the event name. Do not, however, code RESET with TIMER. The system always resets the ECB associated with the TIMER option.

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.

WAIT KEY suspends the issuing task until the enter key or a PF key is pressed. Pressing one of these keys ends the WAIT condition and execution resumes with the instruction following the WAIT KEY. There is no automatic transfer to an

WAIT - Wait for an event to occur *(continued)*

attention routine. The WAIT KEY instruction enqueues the currently active terminal and temporarily inhibits the ATTNLIST capability while the task is suspended by the WAIT instruction.

The key that has been pressed can be identified by the value stored in the second word of the task control block (taskname+2). The program function keys generate values as follows: PF1 generates a value of 1, PF2 generates a value of 2, and so on. The enter key generates a value of 0.

For a 3101 in block mode, pressing the SEND key to satisfy a WAIT KEY will reset changed data tags.

If a READTEXT with TYPE=MODDATA is to be executed after the WAIT KEY, one of the PF keys must be pressed to satisfy the WAIT KEY instruction.

Any terminal I/O operation that takes place as a result of pressing the enter key to satisfy a WAIT KEY instruction will cause a return code to be placed in the first word of the task control block (taskname). If the return code is not a -1, the address of this instruction will be placed in the second word of the task control block (taskname+2). The terminal I/O return codes are described at the end of the PRINTEXT and READTEXT instructions in this manual and also in the *Messages and Codes*.

RESET Reset (clear) the event before waiting. Using RESET will force the wait to occur even if the event has occurred and been posted as complete.

Do not code this operand when you want the system to wait for an event you specified on the EVENT operand of either a PROGRAM or a TASK statement.

P1= Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

WAIT - Wait for an event to occur *(continued)*

Coding Example

The WAIT instruction, at label W1, suspends execution of the primary task until the loaded task, PROG1, signals its completion by posting the ECB labeled LOADECB.

The WAIT instruction at W2 suspends task execution until the operator presses a PF1 key, PF2 key, or the enter key. When one of those keys has been pressed, the task uses the key number, stored in task word 1, to determine what action to take.

The WAIT at label W3 suspends task execution until a 60-second timer has elapsed (it was set by the preceding STIMER instruction).

```
TASK      PROGRAM  BEGIN
LOADECB   ECB
BEGIN     EQU       *
             .
             .
          LOAD     PROG1,EVENT=LOADECB
W1        WAIT     LOADECB
             .
             .
          PRINTEXT '@HIT PF KEY 1 OR 2 TO INDICATE YOUR SELECTION '
W2        WAIT     KEY
          IF       (TASK+2,EQ,1)
             GOTO RTN1
          ELSE
             IF    (TASK+2,EQ,2)
             .
             .
          STIMER   60000
W3        WAIT     TIMER
             .
             .
             .
```

WAITM - Wait for one or more events in a list

The WAITM instruction waits for one or more events to occur from a list of events that you specify with an MECB statement. Up to 20 WAITM operations can be active in the system at any one time.

See "MECB - Create a list of events" on page LR-269 for information on how to code the MECB statement.

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 the WAIT instruction. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for an example that waits for an event to occur in another partition. For more information on cross-partition services, refer to the *Event Driven Executive Language Programming Guide.*

Notes:

1. To use the WAITM instruction, you must have included the SWAITM module in your system and modified the MECBLST keyword on the SYSTEM statement during system generation (See the*Installation and System Generation Guide* for additional information.)

2. The WAITM instruction uses 1024 bytes of storage in partition 1.

3. The system processes the WAITM instruction in the same manner as the WAIT instruction.

```
label         WAITM     mecb,RESET,P1=

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

Operand	*Description*
mecb	The label of the MECB statement that defines the list of events.
RESET	Reset (clear) the events before waiting. Using RESET forces the wait to occur even if the events have occurred and have been posted complete.
P1=	Parameter naming operand. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code this operand.

WAITM - Wait for one or more events in a list *(continued)*

Syntax Example

Wait with reset on a list labeled MECB1.

```
WAITM    MECB1,RESET
  .
  .
  .
```

Post Codes

The following post codes are returned in the first word of the MECB.

Code	Description
X'FFFF'	Successful completion
X'BAD0'	WAITM instruction not supported (SWAITM module not in system)
X'BAD1'	Too many WAITM operations active in system (maximum is 20)
X'BAD2'	Cannot reset MECB because another program is using it
X'BAD3'	Invalid number of events specified

WHERES - Locate an executing program

The WHERES instruction locates another program executing elsewhere in the system. Note that it is not operable with programs you are unable to cancel. These programs are those for which names in storage have been changed. As a result, they do not cancel with the $C command. To locate another program, WHERES searches each partition in ascending order from partition number 1 to determine if the program is contained in that partition. It indicates results of that search by placing a return code in the first word of the task control block. If more than one copy of the program exists, the system reports only the first copy found.

The WHERES instruction does the cross-partition service 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 with an application-defined convention to gain addressability to data or code routines within another program. One such technique is to get 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 also can be used to determine if a particular program is already loaded, thereby avoiding the need to load another copy. See Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 for examples using the WHERES instruction.

Syntax:

```
label         WHERES   progname,address,KEY=,P1=,P2=,P3=

Required:     progname, address
Defaults:     none
Indexable:    none
```

Operand	***Description***
progname	The label of an 8-byte area containing the 1-8 character program name of the program to be located. If the label has fewer than eight characters, the program name must be left-justified and padded with blanks on the right. The program name must begin on a full-word boundary.
address	The label of a word in which the load-point 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.

WHERES - Locate an executing program *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands. P3 is the name of the KEY operand.

Coding Example

The following example demonstrates a use of the cross-partition service WHERES instruction.

$TCBADS is not changed by the WHERES instruction.

```
GETNAME  EQU      *
         READTEXT PGMNAME,'@ENTER THE PROGRAM                            X
               NAME TO BE FOUND'
         IF       (PGMNAME-1,EQ,0,BYTE),GOTO,GETNAME
FINDNAME EQU      *
         WHERES   PGMNAME,ADDRESS,KEY=ADDRKEY    IF THE PROGRAM IS
*                                 FOUND, ADDRESS WILL CONTAIN THE
*                                 ENTRY POINT ADDRESS AND ADDRKEY
*                                 WILL CONTAIN THE ADDRESS KEY
         IF       (TASKNAME,NE,-1),GOTO,NOPGM
         ADD      ADDRKEY,1,RESULT=PARTNUM
         PRINTEXT '@PROGRAM ',SKIP=2
         PRINTEXT PGMNAME
         PRINTEXT ' WAS FOUND IN PARTITION # '
         PRINTNUM PARTNUM
         PRINTEXT ' (ADDRESS SPACE '
         PRINTNUM ADDRKEY
         PRINTEXT ') AT LOAD POINT '
         PRINTNUM ADDRESS
         GOTO     TRYAGAIN
*
NOPGM    EQU      *
         PRINTEXT PGMNAME
         PRINTEXT ' WAS NOT FOUND IN ANY ADDRESS SPACE'
*
TRYAGAIN EQU      *
         PRINTEXT PGMNAME
         QUESTION '@DO YOU WISH TO TRY ANOTHER SEARCH',YES=GETNAME
*
ENDIT    EQU      *
*        GOTO     STOPPER
*        .
PGMNAME  TEXT     LENGTH=8        STORE AREA FOR PROGRAM NAME
ADDRESS  DATA     F'0'            PROGRAM'S PARTITION LOAD POINT
ADDRKEY  DATA     F'0'            ADDRESS SPACE KEY
PARTNUM  DATA     F'0'            PARTITION NUMBER (ADDRKEY + 1)
```

The READTEXT acquires the name of the program for which you are searching. If the Enter key is pressed without typing a response to the READTEXT instruction, the READTEXT and its PROMPT are issued again.

If the program is found, the program name, the address space in which it was located, and the partition number are displayed on the terminal. Otherwise, the system displays a not found message.

WHERES - Locate an executing program *(continued)*

You are always queried by the QUESTION instruction as to whether you wish to try another search. If your reply is no, the program ends. If your reply is yes, the program branches to GETNAME and the program executes again.

Return Codes

Return codes are returned in the first word of the task control block (TCB) of the program or task issuing the instruction. The label of the TCB is the label of your program or task (taskname).

Code	Description
-1	Program found
0	Program not found

WRITE - Write records to a data set

The WRITE instruction transfers one or more records from a a buffer area to a disk, diskette, or tape data set.

You can transfer (write) data sets to disk or diskette either sequentially or directly by relative record. Records are 256 bytes long. The *Operator Commands and Utilities Reference* describes the format of a record created with the text editor of $FSEDIT.

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

The WRITE instruction can take advantage of the cross-partition capability that enables your program to share data with a program or task in another partition. Appendix C, "Communicating with Programs in Other Partitions (Cross-Partition Services)" on page LR-559 contains an example of the cross-partition WRITE operation. You can find more information on cross-partition services in the *Event Driven Executive Language Programming Guide*.

Syntax:

```
label         WRITE    DSx,loc,count,relrecno|blksize,PREC=,
                       END=,ERROR=,WAIT=,P1=,P2=,P3=,P4=

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

Operand ***Description***

DSx The data set to which you are writing. Code DSx, where "x" is a positive integer that indicates the relative position (number) of the data set in the list of data sets you defined on the PROGRAM statement. The value can range from 1 to the maximum number of data sets defined in the list. The maximum range is from 1-9.

You can substitute a DSCB name defined by a DSCB statement for DSx.

loc The label of the buffer area from which data is to be transferred.

WRITE normally assumes the buffer is in the same partition as the currently executing program. You can transfer records from a buffer in another partition, however, by using the cross-partition capability of the WRITE instruction.

count The number of contiguous records you want 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

WRITE - Write records to a data set *(continued)*

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 writes as many records as will fit in the space remaining on the disk data set and returns an end-of-data-set return code to the program.

relrecno The location, by relative record number, where the system is to write a record. The record number is relative to the first record in the data set and the numbering starts with 1. You can code a positive integer or the label of a data area containing the value.

You can request a sequential write operation by coding a 0 or by allowing this operand to default. Sequential WRITE instructions start with relative record 1 or the relative record number specified by a POINT instruction. The supervisor keeps track of sequential WRITE instructions and increments an internal next-record-pointer for each record written in sequential mode (relrecno is 0). Direct WRITE operations (relrecno is not 0) can be intermixed with sequential operations, but this does not change the next-record-pointer used by sequential operations.

If you code a self-defining term for this operand, or an equated value indicated by a plus sign (+), then it is assumed to be a single-word value and is generated as an inline operand. Because this is a one-word value, it is limited to a range of 1 to 32767 (X'7FFF').

If you code an indexable value or an address for this operand, the PREC operand can be used to further define whether relrecno is to be a single-word or double-word value.

If the PREC operand is coded as PREC=D, then the range of relrecno is extended beyond the 32767 value to the limit of a double-word value (2147483647 or X'7FFFFFFF').

blksize The size, in bytes, of the record the system is to write to a tape data set. The range is from 18 to 32767. You can code a self-defining term or the label of a data area containing the value. If you do not code this operand or code a 0, the system uses the default value of 256 bytes.

Do not code this operand in a WRITE instruction containing the relrecno operand.

PREC= This operand further defines the relrecno operand when you specify an address or indexable value for that operand. PREC=S (the default) limits the value of relrecno to single-word precision or to a maximum value of 32767 (X'7FFF').

Coding PREC=D gives the relrecno operand a doubleword precision and extends the range of its maximum value to a doubleword value of 2147483647 (X'7FFFFFFF').

WRITE - Write records to a data set *(continued)*

Do not code this operand in a WRITE instruction containing the blksize operand.

END= The label of the first instruction of the routine to be invoked if an end-of-data-set condition is detected during the WRITE operation (return code=10). If you do not code this operand, the system treats an end-of-data-set (EOD) condition as an error.

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 instruction 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.

Do not code this operand if you code WAIT=NO.

You can set or change the end-of-data by using the SE command of $DISKUT1. See *Operator Commands and Utilities Reference* for additional information.

ERROR= The label of the first instruction of the routine to be invoked if an error condition occurs during the execution of this operation. If you do not code this operand, control passes to the instruction following the WRITE instruction and you must test for any errors.

For tape, if END is not coded, the system treats an EOT as an error and returns 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.

Do not code this operand if you code WAIT=NO

WAIT= YES (the default), to suspend the current task until the operation is complete.

NO, to return control to the current task after the operation is initiated. Your program must issue a subsequent WAIT DSx to determine when the operation is complete.

You cannot code the END and ERROR operands if you code WAIT=NO. You must subsequently test the return code in the Event Control Block (ECB) named DSx or in the first word of the task control block (TCB). The label of the TCB is the label of the program or task (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 an error. For programming purposes, any other return codes should be treated as errors.

WRITE - Write records to a data set *(continued)*

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Special Considerations

If your program is writing data to a diskette and you remove the diskette between write operations and replace it with another diskette, the system writes data to the second diskette before detecting an error.

Syntax Examples for Tape WRITE

1) This WRITE instruction writes a single 1000-byte record from location BUFF1 to a tape data set named OUTDATA. OUTDATA is on a standard-label (SL) tape that has volume serial number 1025.

```
TASK1   PROGRAM   START1,DS=((OUTDATA,1025))
          .
          .
START1 WRITE     DS1,BUFF1,1,1000,ERROR=ERR
```

2) This WRITE instruction writes two records to the tape data set. Each record is 502 bytes in length. Record 1 is located at BUFF2, record 2 is located at BUFF2 + 502 bytes.

```
TASK2   PROGRAM   START2,DS=((OUTDATA,1025))
          .
          .
START2 WRITE     DS1,BUFF2,2,502,ERROR=ERR
```

WRITE - Write records to a data set *(continued)*

Coding Example

The WRITE instruction writes 256 bytes of data, beginning at the location labeled DISKBUFF, into the next sequential record of the first data set specified in the PROGRAM statement. If an end-of-file condition occurs during the write attempt, the program passes control to the label EOFILE. If an unrecoverable I/O error is encountered during the WRITE operation, the program will branch to the DSKWRERR label.

```
SAMPLE    PROGRAM   DS=(CHART1,CHART2)
            .
            .
NXTEMPLY  EQU       *
            .
            .
          MOVEA     #1,DISKBUFF
          MOVE      (000,#1),NAME,(50,BYTE)
          MOVE      (050,#1),STRTADDR,(50,BYTE)
          MOVE      (100,#1),CITY,(50,BYTE)
          MOVE      (150,#1),ZIP,(6,BYTE)
          MOVE      (200,#1),JOBTITLE,(50,BYTE)
          MOVE      (250,#1),JOBDESC,(50,BYTE)
          WRITE     DS1,DISKBUFF,1,0,END=EOFILE,ERROR=DSKWRERR
          GOTO      NXTEMPLY
*
EOFILE    EQU       *
          PRINTEXT '@** EMPLOYEE FILE HAS EXCEEDED AVAILABLE DISK SPACE'
          GOTO      ENDIT
*
DSKWRERR  EQU       *
          PRINTEXT '@UNRECOVERABLE DISK WRITE ERROR ON EMPLOYEE FILE'
          GOTO      ENDIT
          PROGSTOP
DISKBUFF  BUFFER    256,BYTES
          ENDPROG
          END
```

Disk and Tape Return Codes

Disk and 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 to which you are referring.
- The first word of the task control block (TCB). The label of the TCB is the label of your program or task (taskname).

The possible return codes and their meaning for disk and tape are shown in tables later in this section.

If a tape error occurs, the read/write head positions itself immediately following the record in which the error occurred. This indicates that a retry has been attempted but was unsuccessful. The count field, in the WRITE instruction, may or may not have been set to zero under this condition.

WRITE - Write records to a data set *(continued)*

You can get detailed information on an error by using the $LOG utility to capture the I/O error. Refer to the *Problem Determination Guide* for information on how to use $LOG.

Note: If an error is encountered during a sequential I/O operation, the relative record number for the next sequential request is not updated. This will cause errors on all following sequential I/O operations.

Disk/Diskette Return Codes

Return Code	Condition
-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	Read device status I/O instruction error.
5	Unrecoverable I/O error.
6	Error on issuing I/O instruction.
7	A no record found condition occurred, a seek for an alternate sector was performed, and another no record found occurred, for example, no alternate is assigned.
8	A system error occurred while processing an I/O request for a 1024-byte sector diskette.
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	Data set not open or device marked unusable when I/O was requested.
12	DSCB was not OPEN; DDB address = 0.
13	If extended deleted record support was requested ($DCSBFLG bit 3 on), the referenced sector was not formatted at 128 bytes/sector or the request was for more than one 256-byte sector. If extended deleted record support was not requested ($DSCBFLG bit 3 off), a deleted sector was encountered during I/O.
14	The first sector of the requested record was deleted.
15	The second sector of the requested record was deleted.
16	The first and second sectors of the requested record were deleted.
17	Cache fetch error. Contact your IBM customer engineer.
18	Bad cache error. Contact your IBM customer engineer.
24	End of tape.
30	Device not a tape.

WRITE - Write records to a data set *(continued)*

Tape Return Codes and Post Codes

Return Code	Condition
-1	Successful completion.
1	Exception but no status.
2	Error reading cycle steal status.
3	I/O error; retry count exhausted.
4	Error issuing READ CYCLE STEAL STATUS.
6	I/O error issuing I/O operations.
10	End of data; a tape mark was read.
21	Wrong length record.
22	Device not ready.
23	File protected.
24	End of tape.
25	Load point.
26	Unrecoverable I/O error.
27	SL data set not expired.
28	Invalid blocksize.
29	Offline, in-use, or not open.
30	Incorrect device type.
31	Close incorrect address.
32	Block count error during close.
33	Close detected on EOV1.

The following post codes are returned to the event control block (ECB) of the calling program.

Post Code	Condition
-1	Function successful.
101	TAPEID not found.
102	Device not offline.
103	Unexpired data set on tape.
104	Cannot initialize BLP tapes.

WXTRN - Resolve weak external reference symbols

The WXTRN and EXTRN statements identify labels that are not defined within an object module. These labels reside in other object modules that will be link-edited to the module containing the WXTRN or EXTRN statements. The system resolves the reference to an WXTRN or EXTRN label when you link-edit the object module containing the WXTRN or EXTRN statement with the module that defines the label. The module that defines the label must contain an ENTRY statement for that label. (See the ENTRY statement for more information.)

If the system cannot resolve a label during the link-edit, it assigns the label the same address as the beginning of the program. You can include up to 255 WXTRN and EXTRN symbols in your program.

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

Only labels defined by EXTRN statements are used as search arguments during the AUTOCALL processing function of $EDXLINK. Any additional external labels found in the module found by AUTOCALL are used to resolve both WXTRN and EXTRN labels. Refer to the description of $EDXLINK in the *Event Driven Executive Language Programming Guide* for further information.

The main difference between the WXTRN and EXTRN statements is that you must resolve an EXTRN label at link-edit time. It is not necessary to resolve a WXTRN label at link-edit time. The unresolved label coded as an EXTRN receives an error return code from the link process. The same unresolved label coded as a WXTRN receives a warning return code. Both the error and the warning codes indicate unresolved labels. If you know that your application program does not need a label resolved, code it as a WXTRN and your program should execute successfully. Your application will not execute correctly, however, if you try to reference an unresolved label coded in your application program as a WXTRN.

Syntax:

```
blank         WXTRN   label
blank         EXTRN   label

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

Operand	***Description***
label	An external label. You can code up to 10 labels, separated by commas, on a single WXTRN or EXTRN statement.

WXTRN - Resolve weak external reference symbols *(continued)*

Coding Example

The following coding example shows a use of the WXTRN statement.

The labels DATA1, DATA2, LABEL1, and LABEL2 are defined outside this module. The ADD instruction adds the values at DATA1 and DATA2 although the values are defined outside the module where they are being added. The GOTO instructions also can pass control to the the two externally defined labels, LABEL1 and LABEL2.

Each of the external labels could have been entered on a separate line or all three of the EXTRN labels could have been coded on a single EXTRN statement.

```
         .
         .
         .
         EXTRN    DATA1,DATA2
         EXTRN    LABEL1
         WXTRN    LABEL2
         .
         .
         ADD      DATA1,DATA2,RESULT=INDEX
         IF       (INDEX,GT,6)
            GOTO  LABEL1
         ELSE
            GOTO  LABEL2
         ENDIF
         .
         .
INDEX    DATA     F'0'
         .
         .
```

XYPLOT - Draw a curve

The XYPLOT instruction draws a curve that connects points defined by arrays of x and y values. Data values are scaled to screen addresses according to the plot control block. (See the PLOTGIN instruction for a description of the plot control block.) 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
```

Operand	*Description*
x	The label of a data area containing an array of x data values.
y	The label of a data area containing an array of y data values.
pcb	The label of an eight-word plot control block.
n	The label of a data area that contains the number of points to be drawn.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Draw a curve connecting the points specified by an x array at YAXISX and a y array at YAXISY. The data area labeled TWO contains the number of points to be drawn.

```
XYPLOT     YAXISX,YAXISY,PCB,TWO
```

YTPLOT - Draw a curve

The YTPLOT instruction draws a curve connecting points that are equally spaced horizontally and that have heights specified by an array of y values. Data values are scaled to screen addresses according to the plot control block. (See the PLOTGIN instruction for a description of the plot control block.) 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
```

Operand	***Description***
y	The label of a data area containing an array of y data values.
x1	The label of a data area containing the x data value associated with the first point.
pcb	The label of an eight-word plot control block.
n	The label of a data area containing the number of points to be drawn.
inc	The amount of space between points. This operand must be an explicit integer value greater than zero.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a detailed description of how to code these operands.

Syntax Example

Draw a curve with the heights specified by an array of y values at label YDATA. The data area labeled NPTS contains the number of points to be drawn. The instruction leaves one space between each point.

```
         YTPLOT      YDATA,X1,PCB,NPTS,1
```

Appendix A. Formatted Screen Subroutines

You can create and save formatted screen images using the $IMAGE utility. The formatted screen subroutines retrieve and display these images. This appendix describes each of the following subroutines and its operands:

- $IMDATA
- $IMDEFN
- $IMOPEN
- $IMPROT
- $PACK
- $UNPACK.

You can use the formatted screen subroutines with the 4978 and 4979 terminals and with the 3101 terminal in block mode. In addition, by calling these subroutines, you can use screen images created on a 4978 or 4979 terminal on a 3101, and images created on a 3101 terminal on a 4978 or 4979. Refer to the $IMAGE description in *Operator Commands and Utilities Reference* for more information on exchanging terminal screen images.

You must code an EXTRN statement for each subroutine name to which your program refers. You also must link-edit the subroutines with your application program. Specify $AUTO,ASMLIB as the autocall library to include the screen formatting subroutines. Refer to the *Operator Commands and Utilities Reference* for details on the AUTOCALL option of $EDXLINK.

Formatted Screen Subroutines

You call the formatted screen subroutines using the CALL instruction. The following section shows the CALL instruction syntax for each subroutine.

If an error occurs, the terminal I/O return code is in the first word of the task control block (TCB). These errors can come from instructions such as PRINTEXT, READTEXT, and TERMCTRL.

$IMDATA Subroutine

The $IMDATA subroutine displays the initial data values for an image which is in disk storage format. Use $IMDATA:

- To display the unprotected data associated with a screen image, if the buffer contains a screen format retrieved with $IMOPEN.
- To "scatter write" the contents of a user buffer to the input fields of a displayed screen image.

If the buffer is retrieved with $IMOPEN, the buffer begins with either the characters "IMAG" or "IM31" and the buffer index (buffer-4) equals the data length excluding the characters "IMxx."

You can specify a user buffer containing application-generated data. Set the first four bytes of the buffer to USER and set the buffer index (buffer-4) to the data length excluding the characters USER.

All or portions of the screen may be protected after $IMDATA executes. Because the operator cannot key data into protected fields, subsequent read instructions (such as QUESTION, GETVALUE, and READTEXT) should be directed to unprotected areas of the screen, or the protected areas should be erased.

Notes:

1. To use $IMDATA, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.
2. Do not call both $IMDATA and $IMPROT by separate tasks to operate simultaneously. Problems will occur because both call the $IMDTYPE subroutine.

Syntax:

```
label         CALL        $IMDATA,(buffer),(ftab),P2=,P3=

Required:     buffer,ftab  (see note)
Defaults:     none
Indexable:    none
```

Operand	*Description*
buffer	The label of an area containing the image in disk-storage format.

$IMDATA Subroutine *(continued)*

ftab The label of a field table constructed by $IMPROT giving the location (lines,spaces) and size (characters) of each unprotected data field of the image.

Note: The ftab operand is required only if the application executes on a 3101 in block mode or if a user buffer is used in $IMDATA.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMDATA Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
9	Invalid format in buffer

$IMDEFN Subroutine

The $IMDEFN subroutine creates an IOCB for the formatted screen image. You can code the IOCB directly, but the use of $IMDEFN allows the image dimensions to be modified with the $IMAGE utility without requiring a change to the application program. $IMDEFN updates the IOCB to reflect OVFLINE=YES. Refer to the TERMINAL configuration statement in the *Installation and System Generation Guide* for a description of the OVFLINE parameter.

Once you define an IOCB for the static screen, the program can then acquire that screen through ENQT. Once the screen has been acquired, the program can call the $IMPROT subroutine to display the image and the $IMDATA subroutine to display the initial nonprotected fields.

Note: To use $IMDEFN, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

```
label        CALL        $IMDEFN,(iocb),(buffer),topm,leftm,
                         P2=,P3=,P4=,P5=

Required:    iocb,buffer
Defaults:    none
Indexable:   none
```

Operand	*Description*
iocb	The label of an IOCB statement defining a static screen. The IOCB need not specify the TOPM, BOTM, LEFTM, nor RIGHTM parameters; these are "filled in" by the subroutine. The following IOCB statement would normally suffice: `label IOCB SCREEN=STATIC`
buffer	The label of an area containing the screen image in disk storage format. The format is described in the *Event Driven Executive Language Programming Guide*.
topm	This parameter indicates the screen position at which line 0 will appear. If its value is such that lines would be lost at the bottom of the screen, then it is forced to zero. This parameter must equal zero for all 3101 terminal applications. The default is also zero.
leftm	This parameter indicates the screen position at which the left edge of the image will appear. If its value is such that characters would be lost at the right of the screen, then it is forced to zero. This parameter must equal zero for all 3101 terminal applications. The default is also zero.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMDEFN Subroutine *(continued)*

Syntax Example

```
                   CALL      $IMDEFN,(IMGIOCB),(IMGBUFF),0,0
                    .
                    .
                    .
                   ENQT      IMGIOCB
                    .
                    .
                   PROGSTOP
         IMGIOCB   IOCB      SCREEN=STATIC
         IMGBUFF   BUFFER    1024,BYTES
```

$IMOPEN Subroutine

The $IMOPEN subroutine reads a formatted screen image from disk or diskette into your program buffer. You can also perform this operation by using the DSOPEN subroutine or by defining the data set at program load time and issuing the disk READ instruction. Refer to the *Event Driven Executive Language Programming Guide* for a description of buffer sizes. $IMOPEN updates the index word of the buffer with the number of actual bytes read. To refer to the index word, code buffer-4.

Note: To use $IMOPEN, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

```
label        CALL        $IMOPEN,(dsname),(buffer),(type),
                         P2=,P3=,P4=

Required:    dsname,buffer
Defaults:    type=C'4978'
Indexable:   none
```

Operand	*Description*
dsname	The label of a TEXT statement which contains the name of the screen image data set. You can include a volume label, separated from the data set name by a comma.
buffer	The label of a BUFFER statement that defines the storage area into which the image data will be read. Allocate the storage in bytes, as in the following example:

```
label        BUFFER   1024,BYTES
```

type The label of a DATA statement that reserves a 4-byte area of storage and specifies the type of image data set to be read. The data statement must be on a full word boundary. Specify one of the following types:

C'4978'	An image data set with a 4978/4979 terminal format is read. If type is not specified, C'4978' is the default.
C'3101'	An image data set with a 3101 terminal format is read.
C' '	An image data set is read whose format corresponds with the type of terminal enqueued. If neither a 4978/4979 nor 3101 is enqueued (ENQT), a 4978 image format is assumed.

Px= Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a description of these operands.

$IMOPEN Subroutine *(continued)*

$IMOPEN Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
1	Disk I/O error
2	Invalid data set name
3	Data set not found
4	Incorrect header or data set length
5	Input buffer too small
6	Invalid volume name
7	No 3101 image available
8	Data set name longer than eight bytes

$IMPROT Subroutine

The $IMPROT subroutine uses an image created by the $IMAGE utility to prepare the defined protected and blank nonprotected fields for display. At the option of the calling program, a field table can be constructed. The field table gives the location (LINE and SPACES) and length of each unprotected field.

Upon return from $IMPROT, your program can force the protected fields to be displayed by issuing a TERMCTRL DISPLAY. This is not required if a call to $IMDATA follows because $IMDATA forces the display of screen data.

All or portions of the screen may be protected after $IMPROT executes. Because the operator cannot key data into protected fields, subsequent read instructions (such as QUESTION, GETVALUE, and READTEXT) should be directed to unprotected areas of the screen, or the protected areas should be erased.

Notes:

1. To use $IMPROT, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

2. Do not call both $IMPROT and $IMDATA by separate tasks to operate simultaneously. Problems will occur because both call the $IMDTYPE subroutine.

Syntax:

```
label        CALL        $IMPROT,(buffer),(ftab),P2=,P3=

Required:    buffer,ftab  (see note)
Defaults:    none
Indexable:   none
```

Operand	***Description***
buffer	The label of an area containing the screen image in disk storage format. The format is described in the *Event Driven Executive Language Programming Guide*.
ftab	The label of a field table constructed by $IMPROT giving the location (lines, spaces) and size (characters) of each unprotected data field of the image. **Note:** The ftab operand is required only if the application executes on a 3101 in block mode or if a user buffer is used in $IMDATA.
Px=	Parameter naming operands. See "Using The Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMPROT Subroutine *(continued)*

The field table has the following form:

```
label-4         number of fields
label-2         number of words
label          line      * FIELD 1    (one word)
               spaces                 (one word)
               size                   (one word)
label+6        line      * FIELD 2
               spaces
               size
               .
               .
               .
label+6(n-1)   line      * FIELD n
               spaces
               size
```

The field numbers correspond to the following ordering: left to right in the top line, left to right in the second line, and so on to the last field in the last line. Storage for the field table should be allocated with a BUFFER statement specifying the desired number of words using the WORDS parameter. The buffer control word at label-2 is used to limit the amount of field information stored, and the buffer index word at buffer-4 is set with the number of fields for which information was stored, the total number of words being three times that value. If the field table is not desired, code zero for this parameter.

$IMPROT Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
9	Invalid format in buffer
10	Ftab truncated due to insufficient buffer size
11	Error in building ftab from 3101 format; partial ftab created

$PACK Subroutine

The $PACK subroutine moves a byte string and translates it into compressed form.

Note: To use $PACK, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

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

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

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

$PACK Subroutine *(continued)*

Compressed Data Format for $PACK/$UNPACK

F^1	F^2	• • •	Fn	X'00'

Each F^1... Fn is either:

L	C^1	C^2	• • •	Cn

(L is greater than zero and represents the length of chars (C) that follow)

or

L	C

(L is less than zero and represents L repetitions of C)

L and C are one byte in length.

$UNPACK Subroutine

The $UNPACK subroutine moves a byte string and translates it to noncompressed form.

Note: To use $UNPACK, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

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

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

Operand	*Description*
source	The label of a fullword containing the address of a compressed byte string (see Appendix D for the compressed format). At completion of the operation, this parameter is increased by the length of the compressed string.
dest	The label of a fullword containing the address at which the expanded string is to be placed. The length of the expanded string is placed in the byte preceding this location. The $UNPACK subroutine can, therefore, conveniently be used to move and expand a compressed byte string into a TEXT buffer.

For $UNPACK compressed data format see Figure 12 on page LR-550.

Notes

Appendix B. Program Communication Through Virtual Terminals

A "virtual terminal" is a logical EDX device that simulates the actions of a physical terminal. An EDL application program can acquire control of, or enqueue, a virtual terminal just as it would an actual terminal. By using virtual terminals, however, programs can communicate with each other as if they were terminal devices. One program (the primary) loads another program (the secondary) and takes on the role of an operator entering data at a physical terminal. The secondary program can be an application program or a system utility, such as $COPYUT1. You can use virtual terminals, for example, to provide simplified menus for running system utilities. An operator could load a virtual terminal program, select a utility to run, and allow the program to pass predefined parameters to the utility.

Virtual terminals simulate roll screen devices. The terminals communicate through EDL terminal I/O instructions contained in the virtual terminal programs. The programs use a set of virtual terminal return codes to synchronize communication. These return codes are shown under "Virtual Terminal Communication" on page LR-555 and following the READTEXT and PRINTEXT instructions.

Requirements for Defining Virtual Terminals

You must define virtual terminals in pairs. You must include a TERMINAL definition statement for each virtual terminal in your system during system generation. Refer to *Installation and System Generation Guide* for details on how to code the TERMINAL statements for virtual terminals. You must also include the supervisor module IOSVIRT in your system during system generation.

Program Communication Through Virtual Terminals

The DEVICE operand of the TERMINAL statement defines a terminal as a virtual terminal. The ADDRESS operand of the TERMINAL statement contains the label of the other virtual terminal in the pair. The two TERMINAL statements must refer to each other in one of the following ways:

1) The TERMINAL statements below define a pair of virtual terminals. The SYNC=YES operand on the first TERMINAL statement (CDRVTA), indicates that the task enqueuing this virtual terminal will receive the return codes that provide program synchronization.

```
CDRVTA    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,SYNC=YES
CDRVTB    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA
```

2) The TERMINAL statements that follow both contain SYNC=YES. In this case, the task that last attempted an operation will receive a return code for program synchronization.

```
CDRVTA    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,SYNC=YES
CDRVTB    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA,SYNC=YES
```

Considerations for Coding a Virtual Terminal Program

When coding a program that enqueues a virtual terminal you should remember the following:

- The primary virtual terminal program loads the secondary program or system utility with a LOAD instruction.
- The primary virtual terminal program can only communicate with one secondary program or system utility at a time.
- The primary virtual terminal program must include the following COPY statement if you are compiling the program with $EDXASM:

  ```
  COPY    PROGEQU
  ```

- Your program enqueues a virtual terminal with an ENQT instruction. The primary program should enqueue the virtual terminal for the secondary program, load the secondary program, and enqueue a virtual terminal for itself.

  The IOCB statements to which the ENQT instructions refer can be in your primary program or in a secondary application program. The following example shows how a primary program would load the $TERMUT1 utility.

```
                  .
                  .
                  .
                  ENQT     SECOND
                  LOAD     $TERMUT1,LOGMSG=NO,EVENT=ENDWAIT
                  ENQT     PRIMARY
                  .
                  .
                  .
                  PROGSTOP
         PRIMARY  IOCB     CDRVTA      NAME OF THE PRIMARY VIRTUAL TERMINAL
         SECOND   IOCB     CDRVTB      NAME OF THE SECONDARY VIRTUAL TERMINAL
```

Virtual Terminal Communication

To send and receive data through the virtual terminals, application programs use terminal I/O instructions: READTEXT, PRINTEXT, GETVALUE, and PRINTNUM. Virtual terminals do not affect the operation of these instructions. Your program can also generate attention interrupts using TERMCTRL PF, which is described in this book under TERMCTRL (VIRTUAL).

Virtual terminal programs can use a set of return codes to synchronize their operations. Programs or tasks receive the virtual terminal return codes in the first word of their task control block. A program can obtain a return code by referring to the label on the PROGRAM statement.

The virtual terminal return codes and their descriptions follow:

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

Figure 12. Virtual Terminal Return Codes

LINE=nn (X'8Fnn'): Returned for a READTEXT or GETVALUE instruction if the other program issued an instruction with a LINE= operand. This operand tells the system to perform an I/O operation on a certain line of the page or screen. The return code enables the receiving program to reproduce on an actual terminal the output format intended by the sending program.

SKIP=nn (X'8Enn'): The other program issued an instruction with a SKIP= operand. This operand tells the system to skip a number of lines before performing an I/O operation.

Line Received (-2): Indicates that an instruction (usually READTEXT or GETVALUE) has sent information but has not issued a carriage return to move the cursor to the next line. The information is usually a prompt message.

New Line Received (-1): Indicates transmission of a carriage return at the end of the data. The cursor is moved to a new line. This return code and the Line Received return code help programs to preserve the original format of the data they are transmitting.

Not attached (1): A virtual terminal does not or cannot refer to another virtual terminal.

Disconnect (5): The other virtual terminal program ended. This is because you specified a PROGSTOP or an attention list process is complete.

Break (8): Indicates that both virtual terminal programs are attempting to perform the same type of operation. When one program is reading (READTEXT or GETVALUE), the return code means the other program has stopped sending and is waiting for input. When one program is writing, (PRINTEXT or PRINTNUM), the return code means the other program is also attempting to write.

If you defined only one virtual terminal with SYNC=YES, then that task always receives the break code, whether or not it attempted the operation first. If you defined both virtual terminals with SYNC=YES, then the task that last attempted the operation receives the break code.

Sample Virtual Terminal Programs

The sample programs that follow show two types of virtual terminal communication. Both programs assume that the following TERMINAL statements were included during system generation:

```
CDRVTA     TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,SYNC=YES
CDRVTB     TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA
```

1) In this example, the program named SENDER transmits data to the program named RECEIVER. RECEIVER prints the data it received on $SYSPRTR. SENDER is the primary program; RECEIVER is the secondary program.

The SENDER program begins by requesting data from an operator with a READTEXT instruction. SENDER then enqueues the first virtual terminal, loads RECEIVER, and enqueues the second virtual terminal. The DO loop at label CHECK1 issues a READTEXT instruction to determine if RECEIVER is ready to receive data. The instruction

```
READTEXT  LINE,MODE=LINE
```

gets the next line from the RECEIVER program. The loop continues until SENDER receives a return code of 8.

RECEIVER issues a PRINTEXT instruction and then a READTEXT instruction to indicate that it is ready to receive data. When RECEIVER executes the READTEXT, SENDER receives a return code of 8 that indicates both programs are attempting to perform the same operation. SENDER checks the first word of the TCB, finds the return code, exits the DO loop, and executes a PRINTEXT that transmits the operator data to RECEIVER. SENDER then enters a second DO loop at label CHECK2. In this loop, SENDER checks the TCB until it finds a return code of 5. The return code indicates that RECEIVER has printed the data and has completed.

```
SENDER       PROGRAM    START
             PRINT     OFF
             PRINT     ON
A            IOCB       CDRVTA               SYNC TERMINAL
B            IOCB       CDRVTB
START        EQU        *
             READTEXT   DATA,'ENTER DATA TO TRANSMIT ',MODE=LINE
             ENQT       B
             LOAD       RECEIVER,LOGMSG=NO,EVENT=DONE
             ENQT       A
CHECK1       DO         UNTIL,(RC,EQ,8)      DO UNTIL BREAK
              READTEXT    LINE,MODE=LINE
              TCBGET      RC,$TCBCO
             ENDDO
             PRINTEXT   DATA                 SEND INPUT TO OTHER PROGRAM
CHECK2       DO         UNTIL,(RC,EQ,5)      DO UNTIL DISCONNECT
              READTEXT    LINE,MODE=LINE
              TCBGET      RC,$TCBCO
             ENDDO
             WAIT       DONE
             PROGSTOP
DONE         ECB
RC           DATA       F'0'
DATA         TEXT       LENGTH=80
LINE         TEXT       LENGTH=80
             ENDPROG
             END

*****************************************************************

RECEIVER     PROGRAM    START
START        EQU        *
             PRINTEXT   SKIP=1                  SIGNAL TO SEND INPUT
             READTEXT   DATA,MODE=LINE
             ENQT       $SYSPRTR
             PRINTEXT   'THE DATA YOU SENT WAS : '
             PRINTEXT   DATA
             DEQT       $SYSPRTR
             PROGSTOP
DATA         TEXT       LENGTH=80
             ENDPROG
             END
```

2) This example shows how an application can use virtual terminals to process the prompt/reply sequence of the $INITDSK utility. The program initializes volume EDX003.

The replies to $INITDSK prompts begin at label REPLIES+2; each reply is 8 bytes in length (text plus length/count bytes). The program issues a READTEXT until $INITDSK requests input. The program then issues a PRINTEXT to send the reply to the $INITDSK prompt. After $INITDSK ends, the program sends a completion message to the terminal.

```
INIT     PROGRAM  BEGIN
         PRINT    OFF
         PRINT    ON
A        IOCB     A                       SYNC TERMINAL
B        IOCB     B
DEND     ECB
BEGIN    EQU      *
         ENQT     B
         LOAD     $INITDSK,LOGMSG=NO,EVENT=DEND
         ENQT     A                       GET SYNC TERMINAL
         MOVEA    #1,REPLIES+2
         DO       6,TIMES                 REPLY TO PROMPTS
           DO     UNTIL,(RETCODE,EQ,8)    BREAK CODE
             READTEXT  LINE,MODE=LINE     LOOP FOR PROMPT MESSAGES
             TCBGET      RETCODE,$TCBCO   SAVE RETURN CODE
           ENDDO
          PRINTEXT (0,#1)                 SEND REPLY
          ADD      #1,8                   NEXT REPLY
         ENDDO
         READTEXT  LINE,MODE=LINE         PROGRAM END MESSAGE
         WAIT      DEND                   WAIT FOR END EVENT
         DEQT
         PRINTEXT  'EDX003 INITIALIZED'
         PROGSTOP
*
*             DATA  AREA
*
RETCODE  DATA     F'0'                 RETURN CODE
LINE     TEXT     LENGTH=80
REPLIES  EQU      *
         TEXT     'IV     '            COMMAND?
         DATA     CL4' '               4 BYTE FILLER
         TEXT     'EDX003'             VOLUME?
         TEXT     'Y      '            CONTINUE?
         DATA     CL5' '               5 BYTE FILLER
         TEXT     '60     '            NUMBER OF DATA SETS?
         DATA     CL4' '               4 BYTE FILLER
         TEXT     'N      '            VERIFY?
         DATA     CL5' '               5 BYTE FILLER
         DATA     CL5' '               5 BYTE FILLER
         TEXT     'EN     '            COMMAND?
         DATA     CL4' '               4 BYTE FILLER
         ENDPROG
         END
```

Appendix C. Communicating with Programs in Other Partitions (Cross-Partition Services)

EDL programs can communicate with other programs in the system through the use of the following instructions: LOAD, MOVE, STIMER, ATTACH, ENQ, DEQ, WAIT, POST, READ, and WRITE. These instructions enable your program to communicate with another program in the same partition or with a program in another partition. Communication between programs in different partitions is referred to as "cross-partition services".

To communicate with another program, your program must use the WHERES instruction to find the load-point address of the program and the partition where the program resides.

This appendix contains examples of how to communicate with programs in other partitions under the headings:

- "Transferring Data Across Partitions" on page LR-560
- "Starting a Task in Another Partition (ATTACH)" on page LR-566
- "Synchronizing Tasks and the Use of Resources in Different Partitions" on page LR-568

Refer to the *Event Driven Executive Language Programming Guide* for more information on the use of cross-partition services in application programs.

When the system attaches a task, it updates the task control block (TCB) of the task to include the number of the address space where the task is executing. The address space value refers to a partition, and is equal to the partition number minus one. Address space 0, for example, is partition 1. The address space value is also known as the hardware address key. In most of the examples, the system uses the address key and an address your program supplies to provide communication across partitions. The equate that points to the address key in the TCB is $TCBADS.

Note: After issuing a cross-partition service request using $TCBADS, your program should immediately restore $TCBADS to its original value. This procedure can prevent unexpected or unpredictable results such as overlaying other applications with data or having a program wait indefinitely because an ECB was never posted or a DEQ instruction was never issued.

Transferring Data Across Partitions

You can transfer data across partitions using the cross-partition capabilities of the LOAD, MOVE, READ, and WRITE instructions.

Load and Pass Parameters to a Program in Another Partition (LOAD)

In the following example, PROGA loads PROGB into partition 2 and passes PROGB the parameters beginning at the label PROGASW1. After loading PROGB, PROGA waits for the event ENDWAIT, which the system posts when the loaded program ends.

The PARM= operand on PROGB's PROGRAM statement specifies the length of the parameter list that PROGB receives from PROGA. The system recognizes each word in the parameter list by the label $PARMx, where "x" indicates the position of the word in the list. $PARM1 refers to the first word in the list (PROGASW1) and $PARM2 refers to the second word in the list (PROGAKEY).

At the label PROMPT in PROGB, the program displays a prompt message that tells the operator how to cancel PROGB. The MOVEA instruction at label M1 moves the address of CANCELSW into PROGAWRK. The MOVE instruction at label M2 moves the first parameter (the address of PROGASW1) into software register 1. At label M2, PROGB moves the contents of PROGAWRK to the address (0,#1) in PROGA. The TKEY operand of the MOVE instruction supplies the address key of PROGA. PROGB begins a loop at label LOOP until the operator cancels the program.

When the operator presses the attention key and enters "CA", the attention-interrupt-handling routine at label CANCEL in PROGA begins executing. At label M4, the routine moves a value of 1 to the address (0,#1) in PROGB. The TKEY operand on the MOVE instruction supplies the address key for PROGB. The address (0,#1) points to the address of CANCELSW. In PROGB, the IF instruction at label LOOP checks CANCELSW and finds that the variable contains a 1. The instruction passes control to the label STOP and PROGB ends. Control returns to PROGA because the system posts the event ENDWAIT when PROGB ends.

```
PROGA     PROGRAM  START,1,MAIN=YES
COMMAND   ATTNLIST (CA,CANCEL)
CANCEL    EQU      *
          MOVE     #1,PROGASW1
M4        MOVE     (0,#1),1,TKEY=1        CROSS-PARTITION MOVE
          ENDATTN
START     EQU      *
          TCBGET   PROGAKEY,$TCBADS       GET PROGA ADDRESS KEY
*
          LOAD     PROGB,PROGASW1,EVENT=ENDWAIT,LOGMSG=YES,PART=2
          IF       (PROGA,EQ,-1),THEN
            WAIT     ENDWAIT
          ELSE
            PRINTEXT  'LOAD FAILED',SKIP=1
          ENDIF
           .
           .
           .
          PROGSTOP
ENDWAIT   ECB
PROGASW1  DATA    A(PROGASW1)
PROGAKEY  DATA    F'0'
          ENDPROG
          END

***************************************************************

PROGB     PROGRAM START,509,PARM=2
START     EQU      *
           .
           .
           .
PROMPT    PRINTEXT 'TO CANCEL, ENTER: > CA',SKIP=1
          PRINTEXT SKIP=1
M1        MOVEA    PROGAWRK,CANCELSW
M2        MOVE     #1,$PARM1
M3        MOVE     (0,#1),PROGAWRK,TKEY=$PARM2    CROSS-PARTITION MOVE
LOOP      IF       (CANCELSW,EQ,1),GOTO,STOP
          GOTO     LOOP
STOP      EQU      *
          PROGSTOP -1,LOGMSG=NO
PROGAWRK  DATA     F'0'
CANCELSW  DATA     F'0'
          ENDPROG
          END
```

Communicating with Programs in Other Partitions (Cross-Partition Services)

Move Data Across Partitions (MOVE)

The following example shows how to move data to a program in another partition. PROGA finds the program PROGB in storage, stores PROGB's address and address key, and moves data to the dynamic storage area of PROGB.

PROGA uses the WHERES instruction to find the load-point address and address key of PROGB. The WHERES instruction places the load-point address of PROGB in ADDRB and the address key of the program in KEYB.

The READTEXT instruction in PROGA asks the operator to enter up to 30 characters of data. The instruction stores the data in MSG. The MOVE instruction at label M1 moves the address key of PROGB into software register 2. The TCBGET instruction places the address of PROGA's task control block (TCB) in software register 1.

At label M2, the MOVE instruction moves the address of PROGB's dynamic storage area into the data area PROGBBUF in PROGA. The STORAGE= operand on the PROGRAM statement of PROGB causes the system to acquire a 256-byte storage area when it loads the program. The address of this storage area is in PROGB's program header (at $PRGSTG).

At label M3, PROGA saves it's address key in SAVEKEY. The MOVE instruction at M4 moves PROGB's address key to the address key field ($TCBADS) of the TCB. At M5, the MOVE instruction moves the address in PROGB's dynamic storage area to software register 2. PROGA, at M6, moves the data in MSG into PROGB's dynamic storage area. The TKEY operand on the MOVE instruction supplies the address key of PROGB. At M7, PROGA restores its address key from SAVEKEY.

Once PROGB receives the data, it moves the address of the dynamic storage area (contained in $STORAGE) to software register 1. The program moves 30 bytes of data from the dynamic storage area into MSG2, and prints the data it received.

```
PROGA    PROGRAM  START
         COPY     PROGEQU
         COPY     TCBEQU
START    EQU      *
         WHERES   PROGB,ADDRB,KEY=KEYB           FIND PROGB'S LOCATION
         IF       (PROGA,EQ,0),THEN
           PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
           GOTO      DONE
         ENDIF
         READTEXT MSG,'@ENTER UP TO 30 CHARACTERS',MODE=LINE
M1       MOVE     #2,ADDRB
         TCBGET   #1,$TCBVER
M2       MOVE     PROGBBUF,($PRGSTG,#2),FKEY=KEYB
M3       MOVE     SAVEKEY,($TCBADS,#1)             SAVE PROGA'S KEY
M4       MOVE     ($TCBADS,#1),KEYB
M5       MOVE     #2,PROGBBUF
M6       MOVE     (0,#2),MSG,(30,BYTE),TKEY=KEYB
M7       MOVE     ($TCBADS,#1),SAVEKEY             RESTORE PROGA'S KEY
DONE     PROGSTOP
MSG      TEXT     LENGTH=30
PROGBBUF DATA     F'0'
PROGB    DATA     C'PROGB   '
PROGBUF  DATA     F'0'
SAVEKEY  DATA     F'0'
ADDRB    DATA     F'0'
KEYB     DATA     F'0'
         ENDPROG
         END

****************************************************************

PROGB    PROGRAM  START,STORAGE=256
START    EQU      *
           .
           .
           .
         MOVE     #1,$STORAGE              GET STORAGE AREA ADDRESS
         MOVE     MSG2,(0,#1),(30,BYTE)
         PRINTEXT '@THE DATA THAT WAS PASSED IS :'
         PRINTEXT MSG2
         PROGSTOP
MSG2     TEXT     LENGTH=30
         ENDPROG
         END
```

Communicating with Programs in Other Partitions (Cross-Partition Services)

Read Data to or Write Data from a Program in Another Partition

The following example reads data from a data set and stores that data in a buffer in another partition. The data set ACCOUNTS is in PROGA. The buffer is in PROGB. You could use the same coding techniques to write data to a program in another partition (WRITE).

PROGA uses the WHERES instruction to find the load-point address and address key of PROGB. The WHERES instruction places the load-point address of PROGB in ADDRB and the address key of the program in KEYB.

The MOVE instruction at label M1 moves the address key of PROGB into software register 2. The TCBGET instruction places the address of PROGA's task control block (TCB) in software register 1. At label M2, the MOVE instruction moves the address of PROGB's dynamic storage area into PROGBBUF in PROGA. The STORAGE= operand on the PROGRAM statement of PROGB causes the system to acquire a 256-byte storage area when it loads the program. The address of this storage area is in PROGB's program header (at $PRGSTG). At label M3, PROGA saves it's address key in SAVEKEY.

The MOVE instruction at M4 moves PROGB's address key to the address key field ($TCBADS) of the TCB. The READ instruction reads one record from the data set ACCOUNTS into PROGBBUF. Because PROGBBUF is the label of the P2= operand on the READ instruction, the system uses the contents of PROGBBUF as the location where the data is to be stored. After the cross-partition read operation, PROGA restores its address key from SAVEKEY.

Once PROGB receives the data, it moves the address of the dynamic storage area (contained in $STORAGE) to software register 1. The program moves 50 bytes of data from the dynamic storage area into OUTPUT and prints that data.

```
PROGA    PROGRAM  START,DS=ACCOUNTS
         COPY     PROGEQU
         COPY     TCBEQU
START    EQU      *
         WHERES   PROGB,ADDRB,KEY=KEYB           FIND PROGB'S LOCATION
         IF       (PROGA,EQ,0),THEN
           PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
           GOTO      DONE
         ENDIF
M1       MOVE     #2,ADDRB
         TCBGET   #1,$TCBVER
M2       MOVE     PROGBBUF,($PRGSTG,#2),FKEY=KEYB
M3       MOVE     SAVEKEY,($TCBADS,#1)              SAVE PROGA'S KEY
M4       MOVE     ($TCBADS,#1),KEYB
         READ     DS1,*,P2=PROGBBUF                 CROSS-PARTITION READ
         MOVE     ($TCBADS,#1),SAVEKEY              RESTORE PROGA'S KEY
DONE     PROGSTOP
SAVEKEY  DATA     F'0'
PROGB    DATA     C'PROGB   '
ADDRB    DATA     F'0'
KEYB     DATA     F'0'
         ENDPROG
         END

***************************************************************

PROGB    PROGRAM  START,STORAGE=256
START    EQU      *
           .
           .
           .
         MOVE     #1,$STORAGE
         MOVE     OUTPUT,(0,#1),(50,BYTE)
         PRINTEXT '@THE DATA RECEIVED FROM PROGA IS :'
         PRINTEXT OUTPUT,SKIP=1
OUTPUT   TEXT     LENGTH=50
         ENDPROG
         END
```

Communicating with Programs in Other Partitions (Cross-Partition Services)

Starting a Task in Another Partition (ATTACH)

The following example shows how you can use the ATTACH instruction to start, or "attach", a task in another partition. PROGA starts the task labeled TASKADDR in PROGB.

PROGB begins by printing the message "PROGB STARTED". The program then waits for an operator to press the enter key. (This example assumes that the operator will not press the enter key until the task labeled TASKADDR in PROGB has executed.)

PROGA uses the WHERES instruction to find the load-point address and address key of PROGB. The WHERES instruction places the load-point address of PROGB in ADDRB and the address key of the program in KEYB.

The TCBGET instruction places the address of PROGA's task control block (TCB) in software register 1. The MOVE instruction at label M1 saves PROGA's address key. At label M2, the MOVE instruction moves PROGB's address key to the address key field ($TCBADS) of the TCB.

The ADD instruction adds X'34' to the load-point of PROGB. This address points to the first word following PROGB's program header. The ADD instruction places the result of the operation in TASKADDR. Because TASKADDR is the label of the P1= operand on the ATTACH instruction, the system uses the contents of TASKADDR as the address of the task to be attached. After the cross-partition attach operation, PROGA restores its address key from SAVEKEY.

In PROGB, the task labeled TASKADDR is at the first word following the program header generated by the PROGRAM statement. When TASKADDR is attached, it enqueues the system printer, $SYSPRTR, and prints the message "SUBTASK IS ATTACHED". After TASKADDR ends, PROGB waits until an operator presses the enter key.

```
PROGA    PROGRAM  START
         COPY     PROGEQU
         COPY     TCBEQU
START    EQU      *
         WHERES   PROGB,ADDRB,KEY=KEYB         FIND PROGB'S LOCATION
         IF       (PROGA,EQ,0),THEN
           PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
           GOTO      DONE
         ENDIF
         TCBGET   #1,$TCBVER
M1       MOVE     SAVEKEY,($TCBADS,#1)         SAVE PROGA'S KEY
M2       MOVE     ($TCBADS,#1),KEYB
         ADD      ADDRB,X'34',RESULT=TASKADDR  POINT TO TASK ADDRESS
         ATTACH   *,P1=TASKADDR                CROSS-PARTITION ATTACH
M3       MOVE     ($TCBADS,#1),SAVEKEY         RESTORE PROGA'S KEY
           .
           .
           .
DONE     PROGSTOP
SAVEKEY  DATA     F'0'
PROGB    DATA     C'PROGB    '
ADDRB    DATA     F'0'
KEYB     DATA     F'0'
         ENDPROG
         END

*****************************************************

PROGB    PROGRAM  START
********************************************
TASKADDR TASK     NEXT                     *
NEXT     ENQT     $SYSPRTR                 *
         PRINTEXT '@SUBTASK IS ATTACHED'   *
           .                               *
           .                               *
           .                               *
         DEQT                              *
         ENDTASK                           *
********************************************
START    EQU      *
         PRINTEXT '@PROGB STARTED'
         WAIT     KEY
           .
           .
           .
         PROGSTOP
         ENDPROG
         END
```

Communicating with Programs in Other Partitions (Cross-Partition Services)

Synchronizing Tasks and the Use of Resources in Different Partitions

You can synchronize the execution of two or more tasks in different partitions by using the WAIT and POST instructions. The ENQ and DEQ instructions allow you to synchronize the use of a resource by tasks in different partitions.

Post an ECB in Another Partition (POST)

In the following example, PROGA posts an event control block (ECB) in another partition. PROGB contains the ECB that is posted. You could use the same coding techniques to wait for an event in another partition (WAIT).

PROGB begins by waiting for the event labeled ECB1 to be posted. PROGA uses the WHERES instruction to find the load-point address and address key of PROGB. The WHERES instruction places the load-point address of PROGB in ADDRB and the address key of the program in KEYB.

The TCBGET instruction places the address of PROGA's task control block (TCB) in software register 1. The MOVE instruction at label M1 saves PROGA's address key. At label M2, the MOVE instruction moves PROGB's address key to the address key field ($TCBADS) of the TCB.

The ADD instruction adds X'34' to the load-point of PROGB. This address points to the first word following PROGB's program header. The ADD instruction places the result of the operation in PROGBECB. Because PROGBECB is the label of the P1= operand on the POST instruction, the system uses the contents of PROGBECB as the address of the ECB to be posted. After the cross-partition post operation, PROGA restores its address key from SAVEKEY.

In PROGB, the ECB labeled ECB1 is at the first word following the program header generated by the PROGRAM statement. When PROGA posts ECB1, PROGB continues processing.

```
PROGA     PROGRAM  START
          COPY     TCBEQU
START     EQU      *
          WHERES   PROGB,ADDRB,KEY=KEYB     FIND PROGB'S LOCATION
          IF       (PROGA,EQ,0),THEN
            PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
            GOTO      DONE
          ENDIF
          TCBGET   #1,$TCBVER
M1        MOVE     SAVEKEY,($TCBADS,#1)          SAVE PROGA'S KEY
M2        MOVE     ($TCBADS,#1),KEYB
          ADD      ADDRB,X'34',RESULT=PROGBECB   POINT TO PROGB ECB
          POST     *,-1,P1=PROGBECB              CROSS-PARTITION POST
M3        MOVE     ($TCBADS,#1),SAVEKEY          RESTORE PROGA'S KEY
            .
            .
            .
DONE      PROGSTOP
SAVEKEY   DATA     F'0'
PROGB     DATA     C'PROGB    '
ADDRB     DATA     F'0'
KEYB      DATA     F'0'
          ENDPROG
          END

*******************************************************

PROGB     PROGRAM  START
ECB1      ECB
START     EQU      *
          WAIT     ECB1          WAIT FOR ECB1 TO BE POSTED
            .
            .
            .
          PROGSTOP
          ENDPROG
          END
```

Communicating with Programs in Other Partitions (Cross-Partition Services)

Enqueue a Resource in Another Partition (ENQ)

PROGA, in this example, attempts to enqueue a queue control block (QCB) in another partition. The QCB is located in PROGB. PROGA must enqueue the QCB before it can call the subroutine labeled COMMON, which is link-edited to the program. The COMMON subroutine, which is also link-edited to other programs in the system, can only be used by one program at a time.

PROGB begins by waiting for an operator to press the enter key. The program contains the QCB and should remain active while other programs in the system are using the COMMON subroutine.

PROGA uses the WHERES instruction to find the load-point address and address key of PROGB. The WHERES instruction places the load-point address of PROGB in ADDRB and the address key of the program in KEYB. The TCBGET instruction places the address of PROGA's task control block (TCB) in software register 1. The MOVE instruction at label M1 saves PROGA's address key. At label M2, the MOVE instruction moves PROGB's address key to the address key field ($TCBADS) of the TCB.

The ADD instruction adds X'34' to the load-point of PROGB. This address points to the first word following PROGB's program header. The ADD instruction places the result of the operation in PROGBQCB. Because PROGBQCB is the label of the P1= operand on the ENQ instruction, the system uses the contents of PROGBQCB as the address of the QCB to be enqueued.

If the first word of the QCB in PROGB contains a zero, the COMMON subroutine is being used by another program. PROGA, in this case, would pass control to the label CANTHAVE. The busy routine at CANTHAVE would begin by displaying the message "RESOURCE BUSY" and restoring PROGA's address key. If the first word of PROGB's QCB is not a zero, PROGA can call the COMMON subroutine by executing a CALL instruction. When COMMON finishes executing, PROGA dequeues the subroutine. After the cross-partition enqueue operation, PROGA restores its address key from SAVEKEY.

In PROGB, the QCB labeled QCB1 is at the first word following the program header generated by the PROGRAM statement. PROGB remains active until an operator presses the enter key on the terminal.

```
PROGA    PROGRAM  START
         COPY     TCBEQU
         EXTRN    ROUTINE
START    EQU      *
         WHERES   PROGB,ADDRB,KEY=KEYB          FIND PROGB'S LOCATION
         IF       (PROGA,EQ,0),THEN
           PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
           GOTO      DONE
         ENDIF
         TCBGET   #1,$TCBVER
M1       MOVE     SAVEKEY,($TCBADS,#1)          SAVE PROGA'S KEY
M2       MOVE     ($TCBADS,#1),KEYB
         ADD      ADDRB,X'34',RESULT=PROGBQCB   POINT TO PROGB QCB
         ENQ      *,BUSY=CANTHAVE,P1=PROGBQCB   CROSS-PARTITION ENQUEUE
         CALL     ROUTINE
         DEQ
M3       MOVE     ($TCBADS,#1),SAVEKEY
         GOTO     DONE
CANTHAVE EQU      *                             BUSY ROUTINE
         PRINTEXT '@RESOURCE BUSY'
         MOVE     ($TCBADS,#1),SAVEKEY
           .
           .
           .
DONE     PROGSTOP
SAVEKEY  DATA     F'0'
PROGB    DATA     C'PROGB    '
ADDRB    DATA     F'0'
KEYB     DATA     F'0'
         ENDPROG
         END
```

The subroutine link-edited with PROGA looks like:

```
         SUBROUT  ROUTINE
         ENTRY    ROUTINE
         PRINTEXT '@SUBROUTINE HAS BEGUN'
           .
           .
           .
         RETURN
         END

*****************************************************

PROGB    PROGRAM  START
QCB1     QCB
START    EQU      *
         WAIT     KEY
         PROGSTOP
         ENDPROG
         END
```

Notes

Appendix D. EDX Programs, Subroutines, and Inline Code

This appendix describes EDX programs, subroutines, and inline code that you can execute.

EDX Programs

This section describes the following EDX programs:

- $DISKUT3
- $PDS
- $RAMSEC
- $SUBMITP
- $USRLOG.

$DISKUT3 - Manage Data from an Application Program

The $DISKUT3 program enables you to perform the following operations for disks and diskettes from your application program:

- Allocate a data set
- Open a data set
- Delete a data set
- Release unused space in a data set
- Rename a data set
- Set end-of-data indicator in a data set.

You can specify one or more of these operations at the same time. For example, you can open two data sets and allocate two other data sets with one request. Multiple operations save execution time.

You load $DISKUT3 with the LOAD instruction and pass it the address of a list of request block addresses. The request blocks define the operation the system is to perform. This relationship is shown in Figure 13. A word of zeros indicates the end of the request block address list.

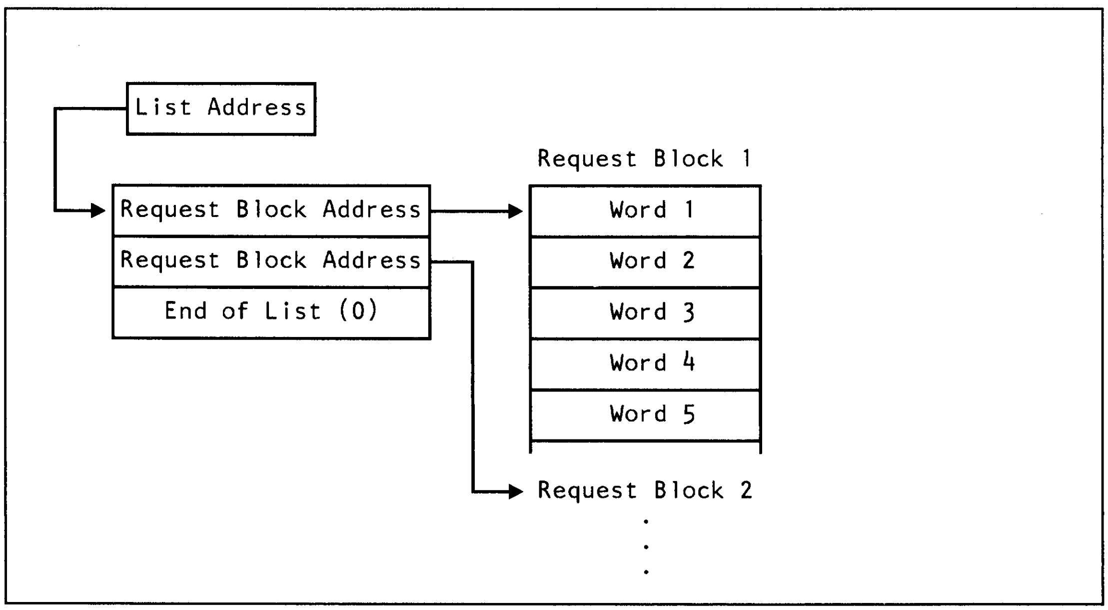

Figure 13. Request Block Example

Request Block Contents

A request block consists of five words as follows:

Word 1: The value in the rightmost byte indicates the operation to be performed. The values are:

Value	Operation
1	Open a data set (OPEN)
2	Allocate a new data set (ALLOCATE)
3	Rename a data set (RENAME)
4	Delete a data set (DELETE)
5	Release unused space in a data set (RELEASE)
6	Set end-of-data indicator in a data set (SETEOD)

The eight leftmost bits are reserved for use as special-purpose flags, as follows:

Bit	Function
00	1 - Indicates that the system should wait if the requested volume is in use. 0 - Indicates that the system should not wait if the requested volume is in use
01	Reserved
02	Reserved
03	Reserved
04	Reserved
05	Reserved
06	Reserved
07	Reserved

For example, if word 1 contains X'8004', the system should delete a data set, but wait if the requested volume is in use.

Word 2: Contains the address of an associated data set control block (DSCB). The DSCB describes the volume and data set you are using. You must specify a DSCB for each operation you request. In addition, you must fill in the data set name ($DSCBNAM) and volume ($DSCBVOL) fields of the DSCB.

Words 3 and 4: The contents of these words vary according to the operation you request. The contents for each operation follows:

Operation	*Contents*
ALLOCATE	Number of records to be allocated (must be in the range of 0 to 2^{31} - 1).
DELETE	Nothing required.
OPEN	Nothing required.
RELEASE	The new size of the data set in records (must be greater than zero and less than the current size.)
RENAME	Word 4 contains the address of a 1-8 byte field containing the new data set name.
SETEOD	Word 4 contains the number of bytes in the last record if it is not yet full; otherwise this word is 0. $DISKUT3 places the value in request block word 4 into bytes 24-25 of the directory member entry (DME). If this value is non-zero, it represents the number of bytes in the last record that is considered not completely full. Bytes 20-23 of the DME are set to the value of $DSCBNEX minus 2. If this value is zero, the last record is considered to be full and bytes 20-23 of the DME are set to the value of $DSCBNEX minus 1.

Word 5: Specifies the data set type. The valid types are:

Code	Type
0	Undefined
1	Data
3	Program
-1	Unspecified

Code 0, 1, or 3 when you allocate a data set. Code -1 when you open, rename, or delete a data set. Upon return from $DISKUT3, the system sets word 5 to 0, 1, or 3, depending upon the type of the data set you specified. If the system sets this word to a value other than -1, $DISKUT3 compares the data set type you specified with the type of the existing data set. If the data sets are not alike, $DISKUT3 returns a return code of 17 and ends.

The system returns the DSCB in an open condition except when it deletes a data set. When you allocate a data set, you do not need to perform an open operation or use DSOPEN.

Special Considerations

Consider the following when using $DISKUT3:

- If you use $DISKUT3 to process data sets that occupy the same volume as your program, you can retrieve the volume name from the $PRGVOL field of the program header. To refer to $PRGVOL, you must include a COPY PROGEQU statement in your program.

- An attempt to delete a data set that does not exist is considered a successful operation.

- An attempt to allocate an existing data set is considered a successful operation if:

  - The existing data set is of the same type as the data set you specified for the operation.

  - The size of the existing data set is the same as size you requested in the operation.

- If you attempt to allocate an existing data set and the data set types match but not the sizes, your program receives a return code indicating whether the data set you requested is smaller or larger than the one that exists.

- The OPEN and SETEOD operations are valid for tape data sets.

EDX Programs *(continued)*

$DISKUT3 Example

The following example uses three of the $DISKUT3 operations (OPEN, ALLOCATE, and RENAME) in an application program.

The LOAD instruction loads $DISKUT3 to open data set (DATA3,) allocate a new data set (DATA4,) and rename an existing data set (DATA1.) DSK3EVNT, the label on the EVENT= operand, is the label of the event control block (ECB) to be posted when $DISKUT3 completes. LISTPTR1 is the label that points to the address of the list of request block addresses. The WAIT instruction waits for the system to post the completion of $DISKUT3.

```
TASK      PROGRAM GO,DS=((DATA1,EDX002),(DATA2,EDX003))
          COPY    DSCBEQU
GO        EQU     *
          .
          .
          .
          LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
          WAIT    DSK3EVNT
          .
          .
          .
          PROGSTOP
          .
          .
DSK3EVNT ECB    0                   SET ECB TO ZERO
LISTPTR1 DC     A(LIST1)            ADDRESS OF LIST OF REQUEST
*                                    BLOCK ADDRESSES
LIST1     DC     A(REQUEST1)
          DC     A(REQUEST2)
          DC     A(REQUEST3)
          DC     F'0'               END OF LIST FLAG
REQUEST1 DC     F'1'               REQUEST:'OPEN' A DATA SET
          DC     A(DSY)             DSCB FOR 'DATA3'
          DC     D'0'               UNUSED FOR OPEN REQUESTS
          DC     F'-1'              UNUSED FOR OPEN REQUESTS
REQUEST2 DC     F'2'               REQUEST: 'ALLOCATE' A DATA SET
          DC     A(DSX)             DSCB FOR 'DATA4'
          DC     D'50'              ALLOCATE 50 RECORDS
          DC     F'1'               DATA SET TYPE IS 'DATA'
REQUEST3 DC     F'3'               REQUEST: 'RENAME' A DATA SET
          DC     A(DS1)             DSCB FOR 'DATA1'
          DC     F'0'               UNUSED FOR RENAME REQUEST
          DC     A(NEWNAME)         ADDRESS OF NEW DATA SET NAME
          DC     F'-1'              FOR RENAME REQUESTS
          DSCB   DS#=DSY,DSNAME=DATA3
          DSCB   DS#=DSX,DSNAME=DATA4
NEWNAME   DC     CL8'RENAMED'       NEW DATA SET NAME
          ENDPROG
          END
```

$DISKUT3 Return Codes

$DISKUT3 return codes are returned to the first word of the data set control block (DSCB). When you specify more than one operation, $DISKUT3 performs the operations in the order you specify. The system returns a return code for each operation attempted.

Note: If you load $DISKUT3 and request more than one operation that refers to the same DSCB, the return code reflects the results of the last operation the system attempted using that DSCB.

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

$PDS - Use Partitioned Data Sets

The display data base utility ($DIUTIL) uses a utility program, $PDS, to make partitioned data sets available for its use. Your programs also can use $PDS to get access to the members of a partitioned data set (such as report data members and realtime data members). You also can use any of the other functions of $PDS in your programs.

Use the LOAD instruction to execute $PDS in your program. $PDS can be used as an overlay program as well as a a program loaded by another program.

$PDS allows you to:

- Open a member
- Allocate a member for a fixed number of records
- Allocate a member for the maximum number of records
- Release unused space from a member
- Delete a member
- Store the next record
- Store a record
- Fetch a record.

The types of members and their member codes are as follows:

Type of member	Member code
Report member	1
Graphic member	2
Graphic member 3D	3
Report data member	4
Plot curve data member	5
Realtime data member	6
Data members you define	7,8,9
You define	10-n

Member types 1, 2, and 3 store commands that are used by $DIINTR to create a display. Member types 4, 5, and 6 contain data that is saved by your application program. Member types 7, 8, and 9 have the same format as member types 4, 5, and 6 but are for use by application programs. Member types 10 and up are for use by application programs.

EDX Programs *(continued)*

Member types 4 through 9 are special members because they contain multiple records with a length of 1 to 32767 bytes. This feature allows the application program to Fetch and Store data by record number within a member. This technique could be used by an application program to update data members defined with the Display Utility Program Set.

You may create members in the following ways:

- Use $DIUTIL utility
  - Data member, member codes 4,5,6
  - User data members, member codes 7,8,9
  - User defined members, member codes 10 and up
  - Member codes 1,2,3 cannot be created by $DIUTIL
- Use $DICOMP program
  - Report member, member code 1
  - Graphic member, member code 2
  - Graphic 3D member, member code 3
- Use $PDS
  - All member types

Allocating a Data Set

A data set that is to be used by $PDS must be allocated using $DISKUT1. Records should be allocated for the directory as well as members. Each record in the directory of a partitioned data set can contain sixteen directory entries except the first record which can contain fifteen. For example, if space is required for 40 members each with five records of space, you should allocate 203 records, 200 for members and three for the directory.

After a data set has been defined by $DISKUT1, it must be formatted for use by $PDS. $DIUTIL functions IN (Initialize), AL (Allocate), and BU (Build Data) are used for this purpose. $PDS can also be used to allocate members. Once members are allocated, they can be used by the application program. The $DIUTIL program is used to maintain the data set.

The data set to be used with $PDS consists of:

- Directory area
- Member area.

Directory Area Format

The first entry in the directory describes the data set and has the following format:

Byte	Usage
0-1	Next available record number for member
2-3	Total size of data set in records
4-5	Number of next directory entry
6-7	Total available directory entries allocated and unallocated
8-15	Unused space

Each succeeding directory entry is 16 bytes with the following format:

Byte	Usage
0-7	EBCDIC member name
8-9	First record number (relative to start of data set)
10-11	Number of records in member
12-13	Member code
14-15	Your code or clear screen indicator

Member Code (bytes 12-13)

-1	Deleted member
0	Available space
1	Report member
2	Graphic member
3	Reserved
4	Report data member
5	Plot curve data member
6	Realtime data member
7-9	Data member you define
10-n	Members you define

Your code (bytes 14-15)

Defined by you and stored by $PDS allocate or a value of 1
if clear screen (ESC,FF) is not to be sent on $DIINTR invocation.

$DIUTIL can be used to display this data for reference.

EDX Programs *(continued)*

Member Area Format

Each member type has a unique format.

Member types 1-3	Display Control Member
No specific format is defined. The data is generated by the $DICOMP Utility Program. See Display Control Member format for information about the content of these members.	

Member Type 4	Report Data Member
Byte	**Usage**
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (1-132)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member Type 5	Plot Curve Data Member
Byte	**Usage**
0-1	X Axis Range
2-3	Y Axis Range
4-5	X Base Line Value
6-7	Y Base Line Value
8-9	Number of records
10-11	Record length in bytes (1-32767)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Note: Each record can be larger than 4 bytes, however relative bytes 0,1 must contain the X-coordinate value and bytes 2,3 must contain the Y-coordinate value.

Member Type 6	Realtime Data Member
Byte	**Usage**
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (must be 16)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member Type 7,8,9	Data Member You Define
Byte	**Usage**
0-7	Unused
8-9	Number of records
10-11	Record length in bytes (1-32767)
12-13	Number of records available
14-15	Unused
16-n	Data Area

Member type 10-n	Member You Define

Display Control Member Format

Each of the display profile elements contained in the control members, type codes (1,2,3), is shown in this section. You may wish to use $PDS to access a control member. The application program could then generate a display profile command string and use $DIINTR to display the results. Following is the format of each of the display profile elements.

LB	Display Characters		
Byte	**Bits**	**Value**	**Content**
0	0-3	1	Display characters code
0	4-7	0	Unused
1	0-7	1-72	Number of characters to display
2-n	0-7	EBCDIC	EBCDIC data to display

MP	Move Position		
Byte	**Bits**	**Value**	**Content**
0	0-3	2	Move Position Code
0-1	4-7/0-7		0-1023 X Coordinate Value
2-3	0-7	0-1023	Y Coordinate Value

For 3D Members:			
Byte	**Bits**	**Value**	**Content**
0	0-3	2	Move Position Code
0-1	4-15	0	Unused
2-3	0-15	-32768 - +32767	X Coordinate Value
4-5	0-15	-32768 - +32767	Y Coordinate Value
6-7	0-15	-32768 - +32767	Z Coordinate Value

EDX Programs *(continued)*

LI **Draw Line**

Byte	Bits	Value	Content
0	0-3	3	Draw Line Code
0-1	4-7/0-7		0-1023 X Coordinate Value
2-3	0-7	0-1023	Y Coordinate Value

For 3D members:

Byte	Bits	Value	Content
0	0-3	3	Move Position Code
0-1	4-15	0	Unused
2-3	0-15	-32768 - +32767	X Coordinate Value
4-5	0-15	-32768 - +32767	Y Coordinate Value
6-7	0-15	-32768 - +32767	Z Coordinate Value

DR **Draw Symbol**

Byte	Bits	Value	Content
0	0-3	4	Draw Symbol Code
0	4-7	1-15	Symbol ID
1	0-7	0-255	Symbol Modifier
2-3	0-7	0-32767	Users Symbol Number
OR			
2	0-5	0	Unused
2	6	0-1	Start top (0) or bottom (1) for Arc
2-3	7/0-7	0-508	# of Y units in Arc

VA **Display Variable**

Byte	Bits	Value	Content
0	0-3	5	Display Variable Code
0	4-7	0-7	Word Number within record
1	0-3	0-15	Function Code
1	4-7	0-3	Type Code
2-3	0-7	1-32767	Record number in Realtime Data Member
4	0-7	1-40	Field Width
5	0-7	0-39	Number of Decimals

EDX Programs *(continued)*

JP	Jump		
Byte	**Bits**	**Value**	**Content**
0	0-3	6	Jump Code
0	4-7	0-7	Word number within record
1	0-7	0-2	Jump Modifier 0=Unconditional 1=Zero 2=Non Zero
2-3	0-7	1-32767	Record number in Realtime Data Member
4-5	0-7	0-32767	Jump to Address (offset in words from beginning of Control Member)

DI	Direct Output to Another Device		
Byte	**Bits**	**Value**	**Content**
0	0-3	8	Direct Output Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character name of output device Refer to ENQT instruction.

PC	Plot Curve from Plot Curve Data Member		
Byte	**Bits**	**Value**	**Content**
0	0-3	9	Plot Curve Code
0	4-7	0	Unused
1	0-7	0 or EBCDIC	EBCDIC character for plot if character plot
2-9	0-7	EBCDIC	8 character member name of a plot data member

**	Display Report Line Items		
Byte	**Bits**	**Value**	**Content**
0	0-3	10	Display Report Line Items
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a report data member

EDX Programs *(continued)*

AD Advance X,Y

Byte	Bits	Value	Content
0	0-3	11	Advance X,Y code
0-1	4-7/0-7	0-1023	X advance value (adjusted by +512)
2-3	0-7	0-1023	Y advance value (adjusted by +512)

For 3D Members:

Byte	Bits	Value	Content
0	0-3	11	Advance X,Y,Z Code
0-1	4-7/0-7	0-1023	X Advance Value (adjusted by +512)
2-3	0-7	0-1023	Y Advance Value (adjusted by +512)
4-5	0-7	0-1023	Z Advance Value (adjusted by +512)

IM Insert Member

Byte	Bits	Value	Content
0	0-3	12	Insert Member Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a central member

LR Draw Line Relative

Byte	Bits	Value	Content
0	0-3	13	Draw Line relative code
0-1	4-7/0-7	0-1023	Delta X Value (adjusted by +512)
2-3	0-7	0-1023	Delta Y Value (adjusted by +512)

EDX Programs *(continued)*

For 3D Members:			
Byte	**Bits**	**Value**	**Content**
0	0-3	13	Draw Line Relative Code
0-1	4-7/0-7	0-1023	Delta X Value (adjusted by +512)
2-3	0-7	0-1023	Delta Y Value (adjusted by +512)
4-5	0-7	0-1023	Delta Z Value (adjusted by +512)

RT	Change Realtime Data Member Name		
Byte	**Bits**	**Value**	**Content**
0	0-3	14	Change Realtime Data Member Code
0	4-7	0	Unused
1	0-7	0	Unused
2-9	0-7	EBCDIC	8 character member name of a new realtime data member (for VA and +P codes)

TD	Display Time and Data		
Byte	**Bits**	**Value**	**Content**
0	0-3	15	Display time and data code
0	4-7	0	Unused
1	0-7	0	Unused

$PDS Example

You get access to $PDS by loading it with the LOAD instruction. The following example shows how to open a member.

```
XYZ       PROGRAM    START, DS=(??)
START     EQU        *
            .
          READTEXT   #MCB,'ENTER MEMBER NAME@'.
            .
          LOAD       $PDS,$MCB,DS=(DS1),EVENT=#PDS,LOGMSG=NO
            .
            .
          WAIT       #PDS
          IF         (#PDS,NE,-1),GOTO,ERROR
            .
            .
 *   NORMAL PROCESSING OF OPENED MEMBER   *
            .
            .
          READ       MBR,BUFF
            .
            .
          WRITE      MBR,BUFF
            .
            .
          PROGSTOP
            .
            .
BUFF      DATA       128F'0'        DISK I/O BUFFER
$MCB      DATA       A(#MCB)        POINTER TO MEMBER CONTROL BLOCK
            .
#MCB      TEXT       LENGTH=8       MEMBER NAME
#MCBCMD   DATA       F'1'           $PDS COMMAND(OPEN)
#MCBDSA   DATA       A(MBR)         ADDRESS OF DSCB
#MCBDT0   DATA       F'0'           Data Field 0
#MCBDT1   DATA       F'0'           Data Field 1
#MCBDT2   DATA       F'0'           Data Field 2
#MCBDT3   DATA       F'0'           Data Field 3
            .
          DSCB       DS#=MBR,DSNAME=DUMMY,VOLSER=DUMMY
            .
          ENDPROG
          END
```

Member Control Block

The 20-byte member control block (MCB) is passed to the $PDS utility program by the PARM facility. The member control block (MCB) is filled in by your application program.

The format of the MCB is as follows:

Byte	Usage
0-7	EBCDIC Member Name
8-9	$PDS Command (see below)
10-11	Address of Callers DSCB
12-19	Data field 0 through 3 (see below)

$PDS Commands (bytes 8-9)

Command	Function
1	Open Member
2	Allocate Member
3	Allocate Member (Maximum Space)
4	Release Space
5	Delete Member
6	Store Next Record
7	Store Record
8	Fetch Record

Command Descriptions

Open Member

The member specified in bytes 0-7 of the MCB is located and the DSCB specified in bytes 10-11 is filled in to point to the member.

Allocate Member

The member specified in bytes 0-7 of the MCB is dynamically allocated with the parameter specified in bytes 14-19.

Allocate Member (maximum space)

The member specified in bytes 0-7 of the MCB is dynamically allocated with the parameter specified in bytes 16-19. Maximum space is allocated.

EDX Programs *(continued)*

Release Space

The member specified in bytes 0-7 of the MCB is located and unused space is returned to the available space in the data set. Bytes 14-15 must contain the number of records that the member will contain.

Delete Member

The member specified in bytes 0-7 of the MCB is located and marked for deletion.

Note: The space occupied by the deleted member is NOT returned to the available space in the data set. Use the utility $DIUTIL to reclaim deleted space.

Store Next Record

The member specified in bytes 0-7 of the MCB is located. The member header is used to determine which record is next and data is stored in that record. Your data buffer address is located in bytes 14-15 of the MCB.

Store Record

The member specified in bytes 0-7 of the MCB is located. The record specified in bytes 12-13 is located and the data is stored in that record. Your data buffer address is located in bytes 14-15 of the MCB.

Fetch Record

The member specified in bytes 0-7 of the MCB is located. The record specified in bytes 12-13 is located. All the data is retrieved and stored in your data buffer. The data buffer address is located in bytes 14-15 of the MCB.

Data fields 0 through 3 must contain modifier information for the various $PDS commands. Also, these areas contain data following the action taken by the $PDS program. The following tables show the data required before executing $PDS and the data returned after $PDS has executed.

Before $PDS Executes:

Command	Data Word 0	Data Word 1	Data Word 2	Data Word 3
Open	N/A	N/A	N/A	N/A
Allocate	N/A	Records	Member Type Code	Your Code
Allocate Max	N/A	N/A	Member Type Code	Your Code
Release	N/A	Records	N/A	N/A
Delete	N/A	N/A	N/A	N/A
Store Next	N/A	Data Buffer Address	N/A	N/A
Store	Record	Data Buffer Address	N/A	N/A
Fetch	Record	Data Buffer Address	N/A	N/A

Note: N/A = Not Applicable

After $PDS Executes:

Command	Data Word 0	Data Word 1	Data Word 2	Data Word 3
Open	1st Record	Records	Member Type Code	Your Code
Allocate	1st Record	Records	Member Type Code	Your Code
Allocate Max	1st Record	Records	Member Type Code	Your Code
Release	N/A	N/A	N/A	N/A
Delete	N/A	N/A	N/A	N/A
Store Next	Record	Data Buffer Address	Records in Member	N/A
Store	Record	Data Buffer Address	Records in Member	N/A
Fetch	Record	Data Buffer	Records	N/A

Note: N/A = Not Applicable

$RAMSEC - Replace Terminal Control Block (4980)

$RAMSEC enables you to replace the current image and/or control stores in the terminal control block (CCB) from an application program by changing the data set names. Replacement data set names are held in the CCB to govern 4980 terminal operations requested after power off and on. They are held until a new $RAMSEC load or IPL occurs.

When you load $RAMSEC from a program, The LOAD instruction passes parameters that indicate the new data set names. You can load your own data sets in combination with any of the two data sets loaded by the initial control store program. The names of the system data sets are:

- Image store: $4980IS0
- Control store: $4980CS0.

In the following data sets, 'x' represents any letter or special character that is allowed in a data set name. The characters 0 through 9 are reserved by EDX. These data sets must appear on the IPL volume. Required names for replacement data sets are:

- Image store: $4980ISx
- Control store: $4980CSx.

Meaning of the Parameter Listings

PARM1	*Meaning*
C'0Y'	When 'Y' is the last character of the image store data set name, the system loads $4980ISY to the terminal. The system modifies the CCB to reflect the current data set.
X'0000'	The system loads $4980IS0, the system default image store, to the terminal. The system modifies the CCB to reflect the current image store data set.
X'0001'	The system loads the image store name currently in the CCB. It does not modify the CCB.
X'FFFF'	The system loads no image store nor does it modify the CCB.

PARM2	*Meaning*
C'0Y'	When 'Y' is the last character of the control store data set name, the system loads $4980ISY to the terminal. The system does not modify the CCB to reflect this data sets name.
X'0000'	The system loads $4980CS0, the system default control store, to the terminal. The system modifies the CCB to reflect the current data set.
X'0001'	The system loads the control store name in the CCB. It does not modify the CCB.
X'FFFF'	The system loads no control store, nor does it modify the CCB.

PARM3	*Meaning*
2F'-1'	Reserved. Must be coded as indicated.

Note: The characters 'X' above indicate hexadecimal numbers. The other character 'Y' in the list above represents any character except the numbers 0 through 9 which are reserved by EDX.

Special Considerations

Consider the following when using $RAMSEC:

- To load a 4980 terminal other than the terminal on which your application is running, you must ENQT the other terminal before loading $RAMSEC.
- Do not specify DEQT=NO on the load instruction, even if you have had to ENQT on a terminal before loading $RAMSEC.
- You cannot replace the default image and control stores at IPL. The system always loads the default image and control stores.

EDX Programs *(continued)*

$RAMSEC Example

The following examples load $RAMSEC to change the image store. In either case, the system loads only the image store, $4980ISY, to the terminal. You can code the parameters as either binary values or characters. Only the rightmost byte, -1, is used by $RAMSEC. The leftmost byte is ignored for all data sets.

```
         MOVE     PARM1+1,C'Y',BYTE   MOVE IN LAST CHAR. OF IMAGE STORE
         LOAD     $RAMSEC,PARM1,EVENT=ECB1,PART=ANY
         WAIT     ECB1               WAIT FOR COMPLETION OF $RAMSEC
           .
           .
           .
PARM1    DC       X'FFFF'            IMAGE STORE PARM
PARM2    DC       X'FFFF'            CONTROL STORE PARM
PARM3    DC       2F'-1'             RESERVED - MUST BE -1
```

Equivalent code would be:

```
         LOAD     $RAMSEC,PARM1,EVENT=ECB1,PART=ANY
         WAIT     label              WAIT FOR COMPLETION OF $RAMSEC
           .
           .
           .
PARM1    DC       C'OY'              IMAGE STORE PARM
PARM2    DC       X'FFFF'            CONTROL STORE PARM
PARM3    DC       2F'-1'             RESERVED - MUST BE -1
```

$RAMSEC Return Codes

A PROGSTOP statement in $RAMSEC issues the following return codes to the application.

Return Code	Condition
-1	Successful operation.
1	Image store load failed.
2	Control store load failed.
3	Image store and control store load failed.
4	PARM3 (two words) was not coded as -1.
5	PARM3 was not coded as -1 and image store load failed.
6	PARM3 was not coded as -1 and control store load failed.
7	PARM3 was not coded as -1 and control store and image store load failed.
8	You did not enqueue 4980.
9	System not able to ENQT 4980 before loading $RAMSEC.

$SUBMITP - Submit a Job for Execution

The $SUBMITP program enables you to submit a job to the job queue processor, $JOBQ, from an application program. You load $SUBMITP from your program with the LOAD instruction and pass it a list of parameters. $SUBMITP can execute two job queue processor commands: SJ and SH. The SJ command submits a job for execution. The SH command submits a job and holds it until you release the job for execution using the RJ command. The RJ command is available under the $SUBMIT utility. (See the *Operator Commands and Utilities Reference* for more information on $SUBMIT.)

You must pass the $SUBMITP program the following parameters (in the order shown):

1. The command name (SJ or SH)
2. The job priority (0-3; 0 is the highest priority)
3. Name of data set containing $JOBUTIL statements
4. Data set volume
5. Address (label) of word containing the job number.

The $SUBMITP program attempts to load the job queue processor if it is not already running. The program places the number of the job at the address of the label you specify in the parameter list.

You must code the EVENT= operand on a LOAD instruction that loads $SUBMITP. The system posts the label on the EVENT= operand when the $SUBMITP program ends. Coding a WAIT instruction following the LOAD instruction enables you to test to see if $SUBMITP submitted the job successfully. You can load $SUBMITP in another partition by specifying the PART= operand on the LOAD instruction.

EDX Programs *(continued)*

$SUBMITP Example

The following example loads $SUBMITP to submit a job for execution.

```
          LOAD    $SUBMITP,PARMS,LOGMSG=NO,EVENT=FINISH
          WAIT    END
          IF      (END,NE,-1),GOTO,ERROR
           .
           .
           .
ERROR     EQU     *
           .
           .
           .
PARMS     EQU     *
          DATA    C'SJ'                  COMMAND NAME
          DATA    X'0002'                JOB PRIORITY
          DATA    CL8'COMPILE'           DATA SET NAME
          DATA    CL6'EDX002'            VOLUME NAME
          DATA    A(JOB)                 ADDRESS OF JOB NUMBER
           .
           .
           .
JOB       DATA    F'0'                   JOB NUMBER RETURNED TO THIS WORD
FINISH    ECB
```

$SUBMITP Return Codes

$SUBMITP return codes are returned to the first word of the event control block you specify with the EVENT= operand of the LOAD instruction.

Code	Condition
-1	Job submitted successfully
1	Job queue is full
2	Invalid data found in job queue data set
3	Disk I/O error is updating queue data set
4	Cannot load $JOBQ
5	Invalid command

$USRLOG - Log Specific Errors From a Program

The USER instruction allows you to use Series/1 assembler code within an EDL program. See "USER - Use assembler code in an EDL program" on page LR-516 for information on use of this instruction. Through this instruction, the $USRLOG subroutine enables you to log specific program errors from an application program. Use of this subroutine is explained below.

Syntax:

```
label        USER     $USRLOG,PARM=(logtype,datatype,
                      dataaddr,datakey,devaddr),
                      P=(logtype,datatype,dataddr,
                      datakey,devaddr)

Required:    logtype,datatype,dataaddr,datakey,devaddr
Defaults:    none
Indexable:   none
```

Operand	***Description***
logtype	The type of log record. Use one of the following values: • 1 — Soft error (device recoverable) • 2 — Hard (unrecoverable) error • 3 — Software recoverable error.
datatype	The type of control block data being logged. Values 0 to 127 are used by the supervisor; values 128 to 255 are reserved for your use. The actual hexadecimal value must be coded.
dataaddr	The address of the log data.
datakey	The address space key of the log data address.
devaddr	The device address.

EDX Programs *(continued)*

$USRLOG Example

The following program example logs a buffer of ones (1s) with $USRLOG.

Define both $DEVLOG and $USRLOG as EXTRNs in programs invoking $USRLOG so as not to incur assembler errors. Also, before executing the $USRLOG subroutine you must link-edit your application program with $$SVC, $$RETURN and $DEVLOG object modules.

```
**********************************************************
*       WHEN LINKING THIS LOG INVOKING PROGRAM USE THE    *
*       FOLLOWING LINK CONTROL                            *
*       AUTOCALL  $AUTO,ASMLIB                            *
*       IN     LOGS,OBJLIB                                *
*       IN     $$SVC,ASMLIB                               *
*       IN     $DEVLOG,ASMLIB                             *
*       LINK   $LOGS,SRCLIB REP END                       *
**********************************************************
          EXTRN     $DEVLOG
          EXTRN     $USRLOG
START     EQU
          TCBGET    ADSO,$TCBADS        GET USER ADDRESS SPACE KEY
          MOVE      ADRSPACE,ADSO       MOVE INTO LOG PARM. LIST
          USER      $USRLOG             LOG RECORD
          PROGSTOP
ERRTYPE1  DC        F'3'                LOGTYPE
DATYPE1   DC        X'0080'             DATATYPE
DATADR1   DC        A(BUFFER)           DATA ADDRESS
ADRSPACE  DC        F'0'                ADDRESS SPACE OF BUFFER
DEVADR1   DC        X'0068'             DEVICE ADDRESS
BUFFER    DC        256C'1'             BUFFER OF ONES
ADSO      DC             F'0'
          ENDPROG
          END
```

To make $USRLOG code reentrant, you may need to disable the system while your program is modifying the parameter list. Note that the logging routine disables the system for a short time. The system is enabled after logging functions are complete. At that time $USRLOG passes control back to the invoking program.

EDX Subroutines

This section describes the following EDX subroutines:

- DSOPEN
- Formatted Screen Subroutines (syntax only)
- Indexed Access Method (syntax only)
- Multiple Terminal Manager (syntax only)
- SETEOD
- UPDTAPE.

You call these subroutines in your application program with the CALL instruction.

The following syntax conventions are used for the subroutines listed in this appendix.

- Operands shown in brackets [ ] are optional
- Operands not shown in brackets are required
- Default values are italicized
- The OR symbol | indicates mutually exclusive operands or parameters.

DSOPEN - Open a data set

You may open a data set from an application program with the DSOPEN copy code. By initializing a DSCB, DSOPEN opens a disk, diskette, or tape data set for input and/or output operations. The results of DSOPEN processing are identical to the implicit open performed by $L or LOAD for data sets specified in the PROGRAM statement.

Note: Only one DSCB can be open to a tape at a time. If a tape has been opened, a close must be issued before another open can be requested.

DSOPEN performs the following functions:

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

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

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

Note: You must code the equates in the order given.

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

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

DSOPEN Error Exit Labels

The DSOPEN subroutine contains labels for a number of error exits. By moving the address of your error routine into the area defined by the DSOPEN label, the subroutine will perform the error routine you supply. The routine you supply can not be another subroutine. If you move a zero into the area defined by the DSOPEN label (except for $$EXIT), the subroutine passes control to the first instruction following the CALL instruction for DSOPEN. The labels are as follows:

Label	*Description*
$DSNFND	Data set name not found in directory If DSOPEN can not find the data set, then it does not fill in the DSCB.
$DSBVOL	Volume not found in disk directory. The system set the DDB pointer in the DSCB to 0 ($DSCBVDE does not equal 0).
$DSIOERR	Read error occurred while DSOPEN was searching the directory. See the READ instruction return codes for more information.
$$EXIT	Exit address. If $$EXIT is 0 and $DSCBNAME equals '$$' or '$$EDXVOL', then DSOPEN initializes the DSCB to the first record (first recond in the library) of the volume specified in the $DSCBVOL. If $$EXIT is 0 and $DSCBNAME is '$$EDXVOL', then DSOPEN initializes the DSCB to the first record of the device where the volume specified on $DSCBVOL resides.
$DSDCEA	Address of area for DSOPEN to store the DCE (Directory Control Entry). This label contains a 0 if this area does not exist.

DSOPEN Considerations

You must have a 256-byte work area labeled DISKBUFR in your program as follows:

```
DISKBUFR  DC    128F'0'
```

The DSCB to be opened can be DS1 to DS9 or a DSCB defined in your program with a DSCB statement. The DSCB must be initialized with a six-character volume name in $DSCBVOL and an eight-character data set name in $DSCBNAM. The volume name can be specified as six blanks, which causes the IPL volume to be searched for the data set.

After DSOPEN processing, #1 contains the number of the directory record containing the member entry and #2 contains the displacement within DISKBUFR to the member entry. The fields $DSCBEND and $DSCBEDB contain the next available logical record data, if any, placed in the directory by SETEOD.

Only one data set on any tape volume may be open at any one time. Multiple data sets, in a program header, or if opened by DSOPEN, cannot refer to more than one data set per tape volume. If this is attempted, the second open attempt will fail and take the Invalid VOLSER error exit.

DSOPEN Example

The following is an example using of the DSOPEN subroutine. The name of the subroutine that calls DSOPEN is USROPEN.

USROPEN opens a data set and returns information about the data set to a 10-word area in the program. Figure 14 on page LR-606 shows the information that USROPEN will return if the DSOPEN subroutine successfully opens the data set.

The call to the USROPEN subroutine would appear as follows:

```
CALL    USROPEN,(label)
```

where (label) is the address of the 10-word area.

At entry to USROPEN, #1 equals A (the DSCB to be opened). This DSCB must have the fields $DSCBNAM and $DSCBVOL filled with the name of the opened data set and the name of the data set volume, respectively.

In order not to receive information about the opened data set after the DSOPEN operation, the call to USROPEN would be coded as follows:

```
CALL    USROPEN,0
```

When USROPEN completes, #1 and #2 are at they were on entry. If DSOPEN takes an error exit during the operation, USROPEN will return the appropriate return code. The return codes set up for USROPEN are as follows:

-1 Operation completed successfully. Data set is open, and if requested, the DM parameters were transferred to a specified area.

2 Data set not found. The data set requested was not found on the volume specified.

3 Volume not found. The volume that the data set is supposed reside on does not exist or is not on line.

6 While DSOPEN was attempting to open the data set, an unrecoverable I/O error occurred on the volume directory.

18 Directory not initialized or is not in correct format.

```
         SUBROUT   USROPEN,OPNDMEP      10-WORD DATA AREA
          .
          .
          .
         MOVE      OPNS#1,#1            SAVE #1
         MOVE      OPNS#2,#2            SAVE #2
*******************************************************************
*  SET UP DSOPEN ERROR EXITS                                      *
*******************************************************************
         MOVEA     $DSNFND,OPNDNF       DATA SET NAME NOT FOUND
         MOVEA     $DSBVOL,OPNVNF       VOLUME NOT FOUND
         MOVEA     $DSIOERR,OPNIOE      ERROR READING DIRECTORY
         MOVEA     $DSBLIB,OPNLIB       VOLUME NOT INITIALIZED
         MOVE      $$EXIT,0             ALLOW $$, $$EDXLIB, $$EDXVOL
*
         CALL      DSOPEN,OPNS#1        CALL DSOPEN
         IF        (OPNDMEP,NE,0)       IF ADDRESS OF DME PARAMETER AREA
*                                        IS PASSED. TRANSFER DM PARAMETER
*                                        INFORMATION FROM DISKBUFFR
           MOVE      #1,OPNDMEP
           MOVE      (0,#1),(DISKBUFR+$$FPMT,#2),8
         ENDIF
OPNXIT   MOVE      #1,0,P2=OPNS#1       RESTORE #1
         MOVE      (0,#1),#2            #2 INTO DSCB
         MOVE      #2,0,P2=OPNS#2       RESTORE #2
         RETURN
*
OPNDNF   MOVE      #2,2                 DATA SET NOT FOUND CODE
         GOTO      OPNXIT               CLEAN UP AND RETURN
*
OPNVNF   MOVE      #2,3                 VOLUME NOT FOUND CODE
         GOTO      OPNXIT               CLEAN UP AND RETURN
*
OPNLIB   MOVE      #2,18                VOLUME NOT INTIALIZED CODE
         GOTO      OPNXIT               CLEAN UP AND RETURN
*
OPNIOE   MOVE      #2,6                 DIRECTORY I/O ERROR CODE
         GOTO      OPNXIT               CLEAN UP AND RETURN
         END
```

EDX Subroutines *(continued)*

After DSOPEN opens the data set, USROPEN fills in the 10-word data area at label OPNDMEP with the following information about the opened data set.

Offset	Contents	
0	DMEKIND	-- Data set type: 0 - Unspecified 1 - Date member (sequential or direct) 3 - Program member
2	DMELA	-- The load address, if the data set is a program (0 - relocatable)
	DMERL	-- The logical record length, if the data set contains data (usually 256).
4	DMEMS	-- If the member is a data set, its size in bytes (doubleword)
	DMEER	-- If the data set contains data, the number of the physical record that contains the last logical record (doubleword)
8	DMEEP	-- If the data set is a program, its entry point.
	DMEEO	-- If the data set contains data, the offset in the EOD physical record of the first byte that is not in a logical record.
10	DMERS	-- If the data set is a program, the size of its relocation dictionary in bytes. This field is reserved if the data set is not a program
12	DMEEOF	-- For data sets containing data, bit 0 equals 1 if DMEER is valid. This field is reserved for programs.

Figure 14. Information Returned from DSOPEN

Formatted Screen Subroutines (Syntax Only)

See Appendix A, "Formatted Screen Subroutines" on page LR-539 for a description of each subroutine and its operands.

All parameters coded in these subroutines must be labels.

```
label     CALL     $IMOPEN,(dsname,volume),(buffer),
                      [(type. C'4978'|C'3101'|C' '),]
                      [P2=,P3=,P4=]

label     CALL     $IMDEFN,(iocb),(buffer)[,topm,leftm,P2=,P3=,P4=]

label     CALL     $IMPROT,(buffer)[,(ftab),P2=,P3=]

label     CALL     $IMDATA,(buffer),(ftab)[,P2=,P3=]

label     CALL     $PACK,source,dest[,P2=,P3=]

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

EDX Subroutines *(continued)*

Indexed Access Method (Syntax Only)

See the IBM Series/1 Event Driven Executive Indexed Access Method (5719-AM3) for a description of each of the following subroutines.

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

label    CALL    IAM,(DISCONN),iacb

label    CALL    IAM,(ENDSEQ),iacb

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

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

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

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

label    CALL    IAM,(PROCESS),iacb,(dscb),(opentab),(mode)

label    CALL    IAM,(PUT|PUTC),iacb,(buff)

label    CALL    IAM,(PUTDE|PUTDEC),iacb,(buff)

label    CALL    IAM,(PUTUP|PUTUPC),iacb,(buff)

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

Multiple Terminal Manager (Syntax Only)

See the *Multiple Terminal Manager Guide and Reference* for a description of each of the following subroutines.

Note: All parameters passed in Multiple Terminal Manager functions must be labels of either values, tables, buffers, or text strings.

```
label    CALL    ACTION,[(buffer),(length),(crlf)]
label    CALL    ASYNCH
label    CALL    BEEP
label    CALL    BLINK
label    CALL    CDATA,(type),(userid),(userclass),(termname),(buffersize)
label    CALL    CHALT
label    CALL    CHGPAN
lable    CALL    CRECVE
label    CALL    CSEND
label    CALL    CYCLE
```

EDX Subroutines *(continued)*

Multiple Terminal Manager (continued)

```
label    CALL    FAN

label    CALL    FILEIO,(FCA),(buffer),(return code)

label    CALL    FTAB,(table),(size),(return code)

label    CALL    GETCUR,(row),(column)

label    CALL    LINK,(pgmname)

label    CALL    LINKON,(pgmname)

label    CALL    MENU

label    CALL    PSEUDO

label    CALL    SETCUR,(row),(column)

label    CALL    SETFMT,(dsname),(rc)

label    CALL    SETPAN,(dsname),(return code)

label    CALL    WRITE,(buffer),(length),(crlf)
```

SETEOD - Set the logical end-of-file on disk

The copy code routine SETEOD allows you to indicate the logical end of file on disk. If your program does not use SETEOD when creating or overwriting a file, the READ end-of-data exception occurs at either the physical or logical end that was set by some previous use of the data set.

SETEOD places the relative record number of the last full physical record in the $$FPMF field of the directory member entry (DME).

Notes:

1. If the $DSCBEDB field is zero, the $$FPMF field is set to the next record pointer field ($DSCBNEX) minus one.

2. If the $DSCBEBD field is not zero, the $$FPMF field is set to the $DSCBNEX minus two.

If the last physical record is partially filled, the number of bytes contained in this record is placed in the $$FPMD of the DME. Otherwise, a zero is placed in this field. (This is done by copying the $DSCBEDB field of the DSCB directly into the DME.) (Further information on the DME can be found in *Internal Design.*)

If the next record pointer field ($DSCBNEX) in the DSCB is 1 when SETEOD is executed, the DME is set to indicate that the data set is empty and $DSCBEND is set to X'-1', indicating that the data set is empty. If $DSCBEOD is zero, the data set is unused.

You can use SETEOD before, during or after any READ or WRITE operation. It does not inhibit further I/O and can be used more than once. The only requirement is that the DSCB passed as input must have been previously opened.

The POINT instruction modifies the $DSCBNEX field. If SETEOD is used after a POINT instruction, the new value of $DSCBNEX is used by SETEOD.

EDX Subroutines *(continued)*

SETEOD requires that the DSOPEN copy code, PROGEQU, TCBEQU, DDBEQU, and DSCBEQU be copied in your program.

To use SETEOD, copy the source code into your program and allocate a work data set as follows:

```
         COPY   TCBEQU
         COPY   PROGEQU
         COPY   DDBEQU
         COPY   DSCBEQU
                 .
                 .
                 .
         COPY   DSOPEN
         COPY   SETEOD
DISKBUFR DC     128F'0'           WORK AREA FOR DSOPEN
```

You invoke SETEOD with the CALL instruction and pass it the DSCB and an I/O error exit routine pointer as parameters. In the following example,

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

DS1 points to a previously opened DSCB and IOERROR is the label of the program routine that receives control if an I/O error occurs.

UPDTAPE - Add Records to a Tape File

The copy code routine UPDTAPE allows you to add records to an existing (or new) tape file. The records added are placed after existing records on the file. On standard label tapes, the routine updates the block count counters in the EOF1 label.

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

```
COPY   UPDTAPE
```

You invoke UPDTAPE with the CALL instruction and pass it the DSCB as a parameter. In the following example,

```
CALL    UPDTAPE,(DS1)
```

DS1 points to a previously opened DSCB.

After the CALL, you must check the return code in the first word of the DSCB for the tape return code. A -1 return code indicates that the tape is positioned correctly for writing records. (See the CONTROL instruction for a list of tape return codes.)

Inline Code (EXTRACT)

This section describes how to find a device type by including the inline copy code routine EXTRACT in your program. EXTRACT determines the device type from the device descriptor block. This routine can be useful for programs that perform operations on a variety of devices. For example, a program may not have to allocate a data set if the data set will reside on a tape. The program can use the EXTRACT routine, in this case, to determine if the device it will use is a tape device.

To use EXTRACT, you must copy the source code into your program. The routine requires the address of a DSCB in #1 and returns the address of a DSCB in #1.

The following example copies the EXTRACT code into the program and checks to see if the device is a tape unit. X'3186' is the device identifier of an IBM 4969 Magnetic Tape unit.

```
MOVEA #1,DS1
COPY  EXTRACT
IF    (#1,EQ,X'3186'),GOTO,TAPEDS
```

Appendix E. Creating, Storing, and Retrieving Program Messages

When designing EDL programs, place prompt messages and other message text in a separate message data set. You save storage space and coding time by doing so. The message utility, $MSGUT1, formats the messages in such a data set. The formatted messages can reside on disk, diskette, or in a module that you link-edit with your application program. The MESSAGE, GETVALUE, READTEXT, and QUESTION instructions enable your program to retrieve and print the appropriate message text when the program executes.

By placing messages in a separate data set, you also can change the text of a message without having to alter and recompile each program that uses that message. For more information on how to build and store program messages, refer to the *Event Driven Executive Language Programming Guide*.

Creating and using your own messages involves the following steps:

1. Creating a data set for source messages

2. Entering the source messages into the data set

3. Formatting and storing the source messages using the message utility, $MSGUT1

4. Retrieving and printing the formatted messages.

The following section covers each of these steps.

Creating, Storing, and Retrieving Program Messages

Creating a Data Set for Source Messages

You create a data set for source messages with one of the text editors described in the *Operator Commands and Utilities Reference*. You can create one or more source message data sets and can store them on any volume. Messages can be simple statements or questions. They can also include any variable fields necessary to contain parameters supplied by your program.

Entering Source Messages into a Data Set

After creating a source message data set, enter your source messages using the following syntax rules:

- Begin each message in column 1.
- Precede each variable field with two *less than* symbols (<<) and follow each variable field with two *greater than* symbols (>>).
- End messages with the characters: /*
- Begin and end comments with double slashes (//comment//). A comment must be associated with a message.
- Use the *at sign* (@) to cause the message to skip to the next line.
- Continue a message on a new line by coding any nonblank character in column 72. Begin the continued line in column 1.

Source messages can be a maximum length of 250 bytes. You can calculate the length of a message by allowing one byte for each character in the text and one byte for each variable field.

The system identifies each message by its position in the source message data set. For example, the system assigns a message number of 3 to the third message in the source message data set. Once you format source messages with the $MSGUT1 utility, add any new messages you have to the end of the source message data set. Leave messages no longer needed in the source message data set or replace them with new messages to preserve the numbering scheme.

Coding Messages with Variable Fields

You may want to construct a message that can return information supplied or generated by your program. To do this, you can code a message with one or more variable fields. When you execute your program, the system inserts the appropriate parameters in these variable fields and prints a complete message. For example, to construct a message that tells a program operator how many records are in a particular data set on a particular volume, code the following:

```
THERE ARE <<SIZE>S> RECORDS IN <<DATA SET NAME>T> ON <<VOLUME>T>/*
```

The variable fields in the previous example are the number of records in the data set (SIZE), the data set name, and the volume name. The variable field names do *not* need to correspond with names in a program.

Note: To print or display a message with variable fields, you must have included the FULLMSG module in your system during system generation.

Set the variable fields off from the message text with two *less than* and two *greater than* symbols (<< >>). The symbols should enclose a description of the field. The system treats the field description as a comment. You can include up to 8 variable fields within a single message.

All variable fields must also contain a *control character* that describes the type of parameter your program will pass to the variable field. The previous example illustrates this point. "S" is the control character in the field <<SIZE>S>; "T" is the control character in the field <<VOLUME>T>. The following is a list of the valid control characters and their descriptions:

C Character data. Specify the number of characters allowed in the field by coding a value from 1 to 250 before the "C" (for example, <<NAME>8C>). There is no default.

T Text. No length is necessary. This control character is similar to "C", but you cannot specify the size of the variable field.

H Hexadecimal data. The length is four EBCDIC characters.

S Single-word integer. Specify a length for the data by coding a value from 1 to 6 before the "S." The default is six EBCDIC characters. The valid range for a single-word integer value is from -32768 to 32767.

D Double-word integer. Specify a length for the data by coding a value from 1 to 11 before the "D." The default is six EBCDIC characters. The valid range for a double-word integer value is from -2147483648 to 2147483647.

Your program passes parameters to a message in the order you specified the parameters in the EDL instruction. The following example shows a MESSAGE instruction with a parameter list (PARMS=):

```
SAMPLE     PROGRAM    START,DS=((MSGSET,EDX003))
              .
              .
           MESSAGE    2,COMP=ID,PARMS=(DSNAME,VOLUME,SIZE)
              .
              .
ID         COMP       'SRCE',DS1,TYPE=DSK
SIZE       DC         F'100'
DSNAME     TEXT       'DATA SET 1'
VOLUME     TEXT       'EDX002'
```

The MESSAGE instruction retrieves message number 2. The source message for message number 2 is:

```
<<DATA SET NAME>T> ON <<VOLUME>T> IS ONLY <<SIZE>S> RECORDS/*
```

When the MESSAGE instruction executes, the system places the first parameter (DSNAME) in the first variable field. It places the second parameter (VOLUME) in the second field, and the third parameter (SIZE) in the third field.

You may, however, want to alter or reword the message in the previous example. It is possible to change the order of variable fields in a source message without changing the order of the parameter list in the program. To do so, code an additional number after the control character. This number, from 1 to 8, points to the parameter that the system should insert into the variable field. The number corresponds to the position of the parameter in the parameter list. For example, <<NAME>C3> tells the system to retrieve the third parameter in the parameter list.

The order of the variable fields in message number 2 has been switched in the following example. Note that a number following the control character, however, points to the correct parameter for the variable field:

```
THERE ARE ONLY <<SIZE>S3> RECORDS IN <<DATA SET NAME>T1> ON          X

<<VOLUME>T2>/*
```

"S3" points to the third parameter in the list (SIZE), "T1" points to the first parameter in the list (DSNAME), and "T2" points to the second parameter in the list (VOLUME).

Sample Source-Message Data Set

The following is a sample of a source-message data set:

```
THIS IS A SAMPLE MESSAGE //THIS IS A SAMPLE COMMENT//  /*
OUTPUT TO SYSTEM PRINTER? /*
ENTER <<TYPE OF VALUE>T1> VALUE LESS THAN <<VALUE>S2> /*
THE PROGRAM HAS PROCESSED THE INPUT DATA./*
ENTER YOUR <<FIRST/LAST/FULL NAME>10C>/*
<<NUMBER>3S> RECORDS HAVE BEEN RECEIVED FROM <<SOURCE>8C>./*
THE ANSWER IS : <<VALUE>D> /*
SORRY, THE DATA YOU ENTERED IS <<ERROR>T>/*
THE DEVICE AT ADDRESS <<DEVICE ADDRESS>H1> IS                     X
IN USE/*
```

Formatting and Storing Source Messages (using $MSGUT1)

Once you have created a source-message data set, you must use the message utility, $MSGUT1, to convert the source messages into a form the system can use. The utility copies the source messages, formats them, and stores the formatted messages. (Refer to the *Operator Commands and Utilities Reference* for a detailed explanation of how to use the message utility.)

You can store the formatted messages on disk or diskette or in a module. If you choose to store your formatted messages in a module, you must link-edit the module containing the messages to your application programs.

Each time you add new messages to the source-message data set, you must reformat the data set with $MSGUT1.

Note: If you included MINMSG in your system during system generation, your program can only retrieve formatted messages from a module.

Retrieving and Printing Formatted Messages

To retrieve a message from storage and include it in your program, you must code a COMP statement and any one of the following instructions: MESSAGE, GETVALUE, QUESTION, and READTEXT. (See the COMP statement and each of the instructions for information on how to retrieve and print formatted messages.)

The system retrieves program messages from the data set or module you allocated with $MSGUT1. If you store formatted messages on disk or diskette, you must include the data set that contains the messages on the PROGRAM statement for your program. The COMP statement must point to this message data set. If you store formatted message in a module, you must link-edit that module to your program. The COMP must also contain the name of this module.

Notes

Appendix F. Conversion Table

The following conversion table shows the hexadecimal, binary, EBCDIC, and ASCII equivalents of decimal values. The table also contains transmission codes for communications devices.

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
0	00	0000 0000	NUL	NUL	NUL		
1	01	0001	SOH	SOH	NUL	space	space
2	02	0010	STX	STX	@	1	1,]
3	03	0011	ETX	ETX	@		
4	04	0100	PF	EOT	space	2	2
5	05	0101	HT	ENQ	space		
6	06	0110	LC	ACK	'		
7	07	0111	DEL	BEL	'	3	
8	08	1000		BS	DLE	4	5
9	09	1001	RLF	HT	DLE		
10	0A	1010	SMM	LF	P		
11	0B	1011	VT	VT	P	5	7
12	0C	1100	FF	FF	0		
13	0D	1101	CR	CR	0	6	6
14	0E	1110	SO	SO	p	7	8
15	0F	1111	SI	SI	p		
16	10	0001 0000	DLE	DLE	BS	8	4
17	11	0001	DC1	DC1	BS		
18	12	0010	DC2	DC2	H		
19	13	0011	TM	DC3	H	9	0
20	14	0100	RES	DC4	(		
21	15	0101	NL	NAK	(	0	Z
22	16	0110	BS	SYN	h	Ⓓ (EOA)	Ⓓ (EOA),9
23	17	0111	IL	ETB	h		
24	18	1000	CAN	CAN	CAN		
25	19	1001	EM	EM	CAN		
26	1A	1010	CC	SUB	X	RS	RS
27	1B	1011	CU1	ESC	X		
28	1C	1100	IFS	FS	8	upper case	upper case
29	ID	1101	IGS	GS	8		⩞
30	1E	1110	IRS	RS	x		
31	1F	1111	IUS	US	x	Ⓒ (EOT)	Ⓒ (EOT)
32	20	0010 0000	DS	space	EOT	@	t
33	21	0001	SOS	!	EOT		
34	22	0010	FS	"	D		
35	23	0011		#	D	/	x
36	24	0100	BYP	$	$		
37	25	0101	LF	%	$	s	n
38	26	0110	ETB	&	d	t	u
39	27	0111	ESC	'	d		
40	28	1000		(	DC4		
41	29	1001		)	DC4	u	e
42	2A	1010	SM	*	T	v	d
43	2B	1011	CU2	+	T		
44	2C	1100		,	4	w	k
45	2D	1101	ENQ	-	4		
46	2E	1110	ACK	.	t		
47	2F	1111	BEL	/	t	x	c
48	30	0011 0000		0	form feed		
49	31	0001		1	form feed	y	l
50	32	0010	SYN	2	L	z	h

*The no-parity TWX code for any given character is the code that has the rightmost bit position off.

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
51	33	0011		3	L		
52	34	0100	PN	4	,		
53	35	0101	RS	5	,		
54	36	0110	UC	6	1	SOA	
55	37	0011 0111	EOT	7	1	Ⓢ (SOA),comma	b
56	38	1000		8	FS		
57	39	1001		9	FS		
58	3A	1010		:	\		
59	3B	1011	CU3	;	\	index	index
60	3C	1100	DC4	<	<		
61	3D	1101	NAK	=	<	Ⓑ (EOB)	
62	3E	1110		>	|		
63	3F	1111	SUB	?	|		
64	40	0100 0000	space	@	STX	Ⓝ (NAK),-	!
65	41	0001		A	STX		
66	42	0010		B	B		
67	43	0011		C	B	i	m
68	44	0100		D	"		
69	45	0101		E	"	k	
70	46	0110		F	b	l	v
71	47	0111		G	b		
72	48	1000		H	DC2		
73	49	1001		I	DC2	m	,
74	4A	1010	¢	J	R	n	r
75	4B	1011	.	K	R		
76	4C	1100	<	L	2	o	i
77	4D	1101	(	M	2		
78	4E	1110	+	N	r		
79	4F	1111	]	O	r	p	a
80	50	0101 0000	&	P	line feed		
81	51	0001		Q	line feed	q	o
82	52	0010		R	J	r	s
83	53	0011		S	J		
84	54	0100		T	*		
85	55	0101		U	*		
86	56	0110		V	j		
87	57	0111		W	j	$	w
88	58	1000		X	SUB		
89	59	1001		Y	SUB		
90	5A	1010	!	Z	Z		
91	5B	1011	$	[	Z	CRLF	CRLF
92	5C	1100	*	\	:		
93	5D	1101	)	]	:	backspace	backspace
94	5E	1110	;	∧	z	idle	idle
95	5F	1111	¬	_	z		
96	60	0110 0000	-	`	ACK		
97	61	0001	/	a	ACK	&	j
98	62	0010		b	F	a	g
99	63	0011		c	F		
100	64	0100		d	&	b	
101	65	0101		e	&		
102	66	0110		f	f		
103	67	0111		g	f	c	f

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
104	68	1000		h	SYN	d	p
105	69	1001		i	SYN		
106	6A	1010	¦	j	V		
107	6B	1011	,	k	V	e	
108	6C	1100	%	l	6		
109	6D	1101		m	6	f	q
110	6E	1110	>	n	v	g	comma
111	6F	1111	?	o	v		
112	70	0111 0000		p	shift out	h	/
113	71	0001		q	shift out		
114	72	0010		r	N		
115	73	0011		s	N	i	y
116	74	0100		t	.		
117	75	0101		u	.		
118	76	0110		v	n	Ⓨ (YAK),period	
119	77	0111		w	n		
120	78	1000		x	RS		
121	79	1001		y	RS		
122	7A	1010	:	z	∧	horiz tab	tab
123	7B	1011	#	{	∧		
124	7C	1100	@	|	>	lower case	lower case
125	7D	1101	'	}	>		
126	7E	1110	=	~	∿		
127	7F	1111	"	DEL	∿	delete	
128	80	1000 0000		NUL	SOH		
129	81	0001	a	SOH	SOH	space	space
130	82	0010	b	STX	A	=	±,[
131	83	0011	c	ETX	A		
132	84	0100	d	EOT	!	<	@
133	85	0101	e	ENQ	!		
134	86	0110	f	ACK	a		
135	87	0111	g	BEL	a	;	#
136	88	1000	h	BS	DC1	:	%
137	89	1001	i	HT	DC1		
138	8A	1010		LF	Q		
139	8B	1011		VT	Q	%	&
140	8C	1100		FF	1		
141	8D	1101		CR	1	'	¢
142	8E	1110		SO	q	>	*
143	8F	1111		SI	q		
144	90	1001 0000		DLE	horiz tab	*	$
145	91	0001	j	DC1	horiz tab		
146	92	0010	k	DC2	l		
147	93	0011	l	DC3	l	(	)
148	94	0100	m	DC4	)		
149	95	0101	n	NAK	)	)	Z
150	96	0110	o	SYN	i	D (EOA),"	(
151	97	0111	p	ETB	i		
152	98	1000	q	CAN	EM		
153	99	1001	r	EM	EM		
154	9A	1010		SUB	Y		
155	9B	1011		ESC	Y		
156	9C	1100		FS	9	upper case	upper case

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
157	9D	1101		GS	9		
158	9E	1110		RS	y		
159	9F	1111		US	y	C (EOT)	C (EOT)
160	A0	1010 0000		Space	ENQ	¢	T
161	A1	0001		!	ENQ		
162	A2	0010	s	"	E		
163	A3	0011	t	#	E	?	X
164	A4	0100	u	$	%		
165	A5	0101	v	%	%	S	N
166	A6	1010 0110	w	&	e	T	U
167	A7	0111	x	'	e		
168	A8	1000	y	(	NAK		
169	A9	1001	z	)	NAK	U	E
170	AA	1010		*	U	V	D
171	AB	1011		+	U		
172	AC	1100		,	5	W	K
173	AD	1101		-	5		
174	AE	1110		.	u		
175	AF	1111		/	u	X	C
176	B0	1011 0000		0	return		
177	B1	0001		1	return	Y	L
178	B2	0010		2	M	Z	H
179	B3	0011		3	M		
180	B4	0100		4	-		
181	B5	0101		5	-		
182	B6	0110		6	m		
183	B7	0111		7	m	(S) (SOA),|	B
184	B8	1000		8	GS		
185	B9	1001		9	GS		
186	BA	1010		:	]		
187	BB	1011		;	]	index	index
188	BC	1100		<	=		
189	BD	1101		=	=	(B) (EOB),ETB	
190	BE	1110		>	{		
191	BF	1111		?	}		
192	C0	1100 0000	{	@	ETX	(N) (NAK),–	
193	C1	0001	A	A	ETX		
194	C2	0010	B	B	C		
195	C3	0011	C	C	C	J	M
196	C4	0100	D	D	#		
197	C5	0101	E	E	#	K	
198	C6	0 0110	F	F	c	L	V
199	C7	0111	G	G	c		
200	C8	1000	H	H	DC3		
201	C9	1001	I	I	DC3	M	"
202	CA	1010		J	S	N	R
203	CB	1011		K	S		
204	CC	1100	⑀	L	3	O	I
205	CD	1101		M	3		
206	CE	1110	⑂	N	s		
207	CF	1111		O	s	P	A
208	D0	1101 0000	}	P	vertical tab		
209	D1	0001	J	Q	vertical tab	Q	O

Conversion Table

Decimal	Hex	Binary	EBCDIC	ASCII (see Notes 1 and 3)	EBASC* (see Notes 2 and 3)	EBCD	CRSP
210	D2	0010	K	R	K	R	S
211	D3	0011	L	S	K		
212	D4	0100	M	T	+		
213	D5	0101	N	U	+		
214	D6	0110	O	V	k		
215	D7	0111	P	W	k	!	W
216	D8	1000	Q	X	ESC		
217	D9	1001	R	Y	ESC		
218	DA	1010		Z	[		
219	DB	1011		[	[	CRLF	CRLF
220	DC	1100		\	;		
221	DD	1101		]	;	backspace	backspace
222	DE	1110		Λ	{	idle	idle
223	DF	1111		—	{		
224	EO	1110 0000	\	`	bell		
225	E1	0001		a	bell	+	J
226	E2	0010	S	b	G	A	G
227	E3	0011	T	c	G		
228	E4	0100	U	d	'	B	+
229	E5	0101	V	e	'		
230	E6	0110	W	f	g		
231	E7	0111	X	g	g	C	F
232	E8	1000	Y	h	ETB	D	P
233	E9	1001	Z	i	ETB		
234	EA	1010		j	W		
235	EB	1011		k	W	E	
236	EC	1100	⑁	l	7		
237	ED	1101		m	7	F	Q
238	EE	1110		n	w	G	comma
239	EF	1111		o	w		
240	FO	1111 0000	0	p	shift in	H	?
241	F1	0001	1	q	shift in		
242	F2	0010	2	r	O		
243	F3	0011	3	s	O	I	Y
244	F4	0100	4	t	/		
245	F5	0101	5	u	/		
246	F6	0110	6	v	o	Ⓨ (YAK), ¬	
247	F7	0111	7	w	o		
248	F8	1000	8	x	US		
249	F9	1001	9	y	US		
250	FA	1010	LVM	z	—	horiz tab	tab
251	FB	1011		{	—		
252	FC	1100		|	?	lower case	lower case
253	FD	1101		}	?		
254	FE	1110		~	DEL		
255	FF	1111		DEL	DEL	delete	

Notes:

1. ASCII terminals attached via #1310, #7850, #2095 with #2096, or #2095 with RPQ D02350.
2. ASCII terminals attached via #1610 or #2091 with #2092.
3. There are two entries for each character, depending on whether the parity is odd or even.

Glossary of Terms and Abbreviations

This glossary defines terms and abbreviations used in the Series/1 Event Driven Executive software publications. All software and hardware terms pertain to EDX. This glossary also serves as a supplement to the *IBM Data Processing Glossary*, GC20-1699.

$SYSLOGA, $SYSLOGB. The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG. The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR. The name of the system printer.

abend. Abnormal end-of-task. Termination of a task prior to its completion because of an error condition that cannot be resolved by recovery facilities while the task is executing.

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 key. The key on the display terminal keyboard that, if pressed, tells the operating system that you are entering a command.

attention list. A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup. A copy of data to be used in the event the original data is lost or damaged.

base record slots. Space in an indexed file that is reserved for based records to be placed.

base records. Records are placed into an indexed file while in load mode or inserted in process mode with a new high key.

basic exchange format. A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB). A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

Glossary of Terms and Abbreviations

block. (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

block mode. The transmission mode in which the 3101 Display Station transmits a data data stream, which has been edited and stored, when the SEND key is pressed.

BSCAM. See binary synchronous communications access method.

binary synchronous communications access method. A form of binary synchronous I/O control used by the Series/1 to perform data communications between local or remote stations.

BSCDDB. See binary synchronous device data block.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing. Access of a tape without any label processing support.

CCB. See terminal control block.

central buffer. The buffer used by the Indexed Access Method for all transfers of information between main storage and indexed files.

character image. An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table. An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

character mode. The transmission mode in which the 3101 Display Station immediately sends a character when a keyboard key is pressed.

cluster. In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction). A character used with ACCA terminal to indicate a reverse in the direction of data movement.

cold start. Starting the spool facility by erasing any spooled jobs remaining in the spool data set from any previous spool session.

command. A character string from a source external to the system that represents a request for action by the system.

common area. A user-defined data area that is mapped into the partitions specified on the SYSTEM definition statement. 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.

constant. A value or address that remains unchanged thoughout program execution.

controller. A device that has the capability of configuring the GPIB bus by designating which devices are active, which devices are listeners, and which device is the talker. In Series/1 GPIB implementation, the Series/1 is always the controller.

conversion. See update.

control station. In BSCAM communications, the station that supervises a multipoint connection, and performs polling and selection of its tributary stations. The status of control station is assigned to a BSC line during system generation.

cross-partition service. A function that accesses data in two partitions.

cross-partition supervisor. A supervisor in which one or more supervisor modules reside outside of partition 1 (address space 0).

data block. In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data record. In an indexed file, the records containing customer data.

data set. A group of records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB). A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down. An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE. See directory control entry.

device data block (DDB). A control block that describes a disk or diskette volume.

direct access. (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation. (3) A condition in terminal I/O where a READTEXT or a PRINTEXT is directed to a buffer which was previously enqueued upon by an IOCB.

directory. (1) A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free space. (2) A series of contiguous records on a device that describe the contents in terms of allocated volumes and free space. (3) For the Indexed Access Method Version 2, a data set that defines the relationship between primary and secondary indexed files (secondary index support).

directory control entry (DCE). The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME). A 32-byte directory entry describing an allocated data set or volume.

display station. An IBM 4978, 4979, or 3101 display terminal or similar terminal with a keyboard and a video display.

DME. See directory member entry.

DSCB. See data set control block.

dynamic storage. An increment of storage that is appended to a program when it is loaded.

end-of-data indicator. A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB. See event control block.

EDL. See Event Driven Language.

emulator. The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT). A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

enter key. The key on the display terminal keyboard that, if pressed, tells the operating system to read the information you entered.

event control block (ECB). A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

Event Driven Language (EDL). The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output). An EDL facility that provides user controlled access to Series/1 input/output devices.

external label. A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN). The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA. See file control area.

FCB. See file control block.

file. A set of related records treated as a logical unit. Although file is often used interchangeably with data set, it usually refers to an indexed or a sequential data set.

file control area (FCA). A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB). The first block of an indexed file. It contains descriptive information about the data contained in the file.

file control block extension. The second block of an indexed file. It contains the file definition parameters used to define the file.

file manager. A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

floating point. A positive or negative number that can have a decimal point.

formatted screen image. A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool. In an indexed data set, a group of blocks that can be used for either data blocks or index blocks. These differ from other free blocks in that these are not initially assigned to specific logical positions in the file.

free space. In an indexed file, records blocks that do not currently contain data, and are available for use.

free space entry (FSE). An 8-byte directory entry defining an area of free space within a volume or a device.

FSE. See free space entry.

general purpose interface bus. The IEEE Standard 488-1975 that allows various interconnected devices to be attached to the GPIB adapter (RPQ D02118).

Glossary of Terms and Abbreviations

GPIB. See general purpose interface bus.

group. A unit of 100 records in the spool data set allocated to a spool job.

H exchange format. A standard format for exchanging data on diskettes between systems or devices.

host assembler. The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system. Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB. See indexed access control block.

IAR. See instruction address register.

ICB. See indexed access control block.

IIB. See interrupt information byte.

image store. The area in a 4978 that contains the character image table.

immediate data. A self-defining term used as the operand of an instruction. It consists of numbers, messages or values which are processed directly by the computer and which do not serve as addresses or pointers to other data in storage.

index. In an indexed file, an ordered collection of pairs of keys and pointers, used to sequence and locate records.

index block. In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB). The control block that relates an application program to an indexed file.

indexed access method. An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set. Synonym for indexed file.

indexed file. A file specifically created, formatted and used by the Indexed Access Method. An indexed file is sometimes called an indexed data set.

index entry. In an indexed file, a key-pointer pair, where the pointer is used to locate a lower-level index block or a data block.

index register (#1, #2). Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB). A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer, or an optional reference to a user provided buffer.

instruction address register (IAR). The pointer that identifies the machine 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).

integer. A positive or negative number that has no decimal point.

interactive. The mode in which a program conducts a continuous dialogue between the user and the system.

internal label. An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB). In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

invoke. To load and activate a program, utility, procedure, or subroutine into storage so it can run.

job. A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement. A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor. The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

jumper. (1) A wire or pair of wires which are used for the arbitrary connection between two circuits or pins in an attachment card. (2) To connect wire(s) to an attachment card or to connect two circuits.

key. In the Indexed Access Method, one or more consecutive characters used to identify a record and establish its order with respect to other records. See also key field.

key field. A field, located in the same position in each record of an indexed file, whose content is used for the key of a record.

level status block (LSB). A Series/1 hardware data area that contains processor status. This area is eleven words in length.

library. A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line. A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit. The process of resolving external symbols in one or more object modules. A link edit is performed with $EDXLINK whose output is a loadable program.

listener. A controller or active device on a GPIB bus that is configured to accept information from the bus.

load mode. In the Indexed Access Method, the mode in which records are loaded into base record slots in an indexed file.

load module. A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point. (1) Address in the partition where a program is loaded. (2) A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock. In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

logical screen. A screen defined by margin settings, such as the TOPM, BOTM, LEFTM and RIGHTM parameters of the TERMINAL or IOCB statement.

LSB. See level status block.

mapped storage. The processor storage that you defined on the SYSTEM statement during system generation.

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.

message. In data communications, the data sent from one station to another in a single transmission. Stations communication with a series of exchanged messages.

multifile volume. A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager. An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file. A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

new high key. A key higher than any other key in an indexed file.

nonlabeled tapes. Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character. A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu. A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay. The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program or by calling overlay segments that have been specified in the OVERLAY statement of $EDXLINK.

overlay area. A storage area within a program reserved for overlay programs specified in the PROGRAM statement or overlay segments specified in the OVERLAY statement in $EDXLINK.

overlay program. A program in which certain control sections can use the same storage location at different times during execution. An overlay program can execute concurrently as an asynchronous task with other programs and is specified in the EDL PROGRAM statement in the main program.

overlay segment. A self-contained portion of a program that is called and sequentially executes as a synchronous task. The entire program that calls the overlay segment need not be maintained in storage while the overlay segment is executing. An overlay segment is specified in the OVERLAY statement of $EDXLINK or $XPSLINK (for initialization modules).

overlay segment area. A storage area within a program or supervisor reserved for overlay segments. An overlay segment area is specified with the OVLAREA statement of $EDXLINK.

parameter selection menu. A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition. A contiguous fixed-sized area of storage. Each partition is a separate address space.

performance volume. A volume whose name is specified on the DISK definition statement so that its address is found during IPL, increasing system performance when a program accesses the volume.

physical timer. Synonym for timer (hardware).

polling. In data communications, the process by which a multipoint control station asks a tributary if it can receive messages.

precision. The number of words in storage needed to contain a value in an operation.

prefind. To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary file. An indexed file containing the data records and primary index.

primary file entry. For the Indexed Access Method Version 2, an entry in the directory describing a primary file.

primary index. The index portion of a primary file. This is used to access data records when the primary key is specified.

primary key. In an indexed file, the key used to uniquely identify a data record.

primary-level index block. In an indexed file, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu. The program selection screen displayed by the Multiple Terminal Manager.

primary option menu. The first full screen display provided by the Session Manager.

primary station. In a Series/1-to-Series/1 Attachment, the processor that controls communication between the two computers. Contrast with secondary station.

primary task. The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority. A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode. In the Indexed Access Method, the mode in which records can be retrieved, updated, inserted, or deleted.

processor status word (PSW). A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program. A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header. The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager. A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field. A field in which the operator cannot use the keyboard to enter, modify, or erase data.

PSW. See processor status word.

QCB. See queue control block.

QD. See queue descriptor.

QE. See queue element.

queue control block (QCB). A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD). A control block describing a queue built by the DEFINEQ instruction.

queue element (QE). An entry in the queue defined by the queue descriptor.

quiesce. To bring a device or a system to a halt by rejection of new requests for work.

quiesce protocol. A method of communication in one direction at a time. When sending node wants to receive, it releases the other node from its quiesced state.

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. (3) In BSCAM communications, the portions of data transmitted in a message. Record length (and, therefore, message length) can be variable.

recovery. The use of backup data to re-create 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 block address (RBA). The location of a block of data on a 4967 disk relative to the start of the device.

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.

relocation dictionary (RLD). The part of an object module or load module that is used to identify address and name constants that must be adjusted by the relocating loader.

remote management utility control block (RCB). A control block that provides information for the execution of remote management utility functions.

reorganize. The process of copying the data in an indexed file to another indexed file in a manner that rearranges the data for more optimum processing and free space distribution.

restart. Starting the spool facility w the spool data set contains jobs from a previous session. The jobs in the spool data set can be either deleted or printed when the spool facility is restarted.

return code. An indicator that reflects the results of the execution of an instruction or subroutine. The return code is usually placed in the task code word (at the beginning of the task control block).

roll screen. A display screen which is logically segmented into an optional history area and a work area. Output directed to the screen starts display at the beginning of the work area and continues on down in a line-by-line sequence. When the work area gets full, the operator presses ENTER/SEND and its contents are shifted into the optional history area and the work area itself is erased. Output now starts again at the beginning of the work area.

SBIOCB. See sensor based I/O control block.

second-level index block. In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary file. See secondary index.

secondary index. For the Indexed Access Method Version 2, an indexed file used to access data records by their secondary keys. Sometimes called a secondary file.

secondary index entry. For the Indexed Access Method Version 2, this an an entry in the directory describing a secondary index.

secondary key. For the Indexed Access Method Version 2, the key used to uniquely identify a data record.

secondary option menu. In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task. Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

secondary station. In a Series/1-to-Series/1 Attachment, the processor that is under the control of the primary station.

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.

selection. In data communications, the process by which the multipoint control station asks a tributary station if it is ready to send messages.

self-defining term. A decimal, integer, or character that the computer treats as a decimal, integer, or character and not as an address or pointer to data in storage.

sensor based I/O control block (SBIOCB). A control block containing information related to sensor I/O operations.

sequential access. The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR). A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

service request. A device generated signal used to inform the GPIB controller that service is required by the issuing device.

session manager. A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource. A resource that can be used by more than one task at the same time.

Glossary of Terms and Abbreviations

shut down. See data set shut down.

source module/program. A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

spool job. The set of print records generated by a program (including any overlays) while engueued to a printer designated as a spool device.

spool session. An invocation and termination of the spool facility.

spooling. The reading of input data streams and the writing of output data streams on storage devices, concurrently with job execution, in a format convenient for later processing or output operations.

SRQ. See service request.

stand-alone dump. An image of processor storage written to a diskette.

stand-alone dump diskette. A diskette supplied by IBM or created by the $DASDI utility.

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.

station. In BSCAM communications, a BSC line attached to the Series/1 and functioning in a point-to-point or multipoint connection. Also, any other terminal or processor with which the Series/1 communicates.

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 defining I/O devices and selecting software options to create a supervisor tailored to the needs of a specific Series/1 hardware configuration and application.

system partition. The partition that contains the root segment of the supervisor (partition number 1, address space 0).

talker. A controller or active device on a GPIB bus that is configured to be the source of information (the sender) on the bus.

tape device data block (TDB). A resident supervisor control block which describes a tape volume.

tapemark. A control character recorded on tape used to separate files.

task. The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task are secondary tasks.

task code word. The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB). A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor. The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB. See task control block.

terminal. A physical device defined to the EDX system using the TERMINAL configuration statement. EDX terminals include directly attached IBM displays, printers and devices that communicate with the Series/1 in an asynchronous manner.

terminal control block (CCB). A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB). A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager. The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server. A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

terminal support. The support provided by EDX to manage and control terminals. See terminal.

timer. The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card (4955 only) or the native timer (4952, 4954, and 4956). Only one or the other is supported by the Event Driven Executive.

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.

tributary station. In BSCAM communications, the stations under the supervision of a control station in a multipoint connection. They respond to the control station's polling and selection.

unmapped storage. The processor storage in your processor that you did not define on the SYSTEM statement during system generation.

unprotected field. A field in which the operator can use the keyboard to enter, modify or erase data. Also called non-protected field.

update. (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit. (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a user written routine can be given control.

variable. An area in storage, referred to by a label, that can contain any value during program execution.

vary offline. (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is unknown by the system.

vary online. To place a device in a state where it is available for use by the system.

vector. An ordered set or string of numbers.

volume. A disk, diskette, or tape subdivision defined using $INITDSK or $TAPEUT1.

volume descriptor entry (VDE). A resident supervisor control block that describes a volume on a disk or diskette.

volume label. A label that uniquely identifies a single unit of storage media.

Index

The following index contains entries for this book only. See the *Library Guide and Common Index* for a Common Index to all Event Driven Executive books.

Special Characters

Index

C

Index

E

G

Index

Index

Index

S

Index

T

U

Index

IBM Series/1 Event Driven Executive

Publications Order Form

Instructions:

1. Complete the order form, supplying all of the requested information. (Please print or type.)
2. If you are placing the order by phone, dial **1-800-IBM-2468.**
3. If you are mailing your order, fold the order form as indicated, seal with tape, and mail. We pay the postage.

Ship to:

Name:

Address:

City:

State: Zip:

Bill to:

Customer number:

Name:

Address:

City:

State: Zip:

Your Purchase Order No.:

Phone: ( )

Signature:

Date:

Order:

Description	Order number	Qty.
Reference books:		
Set of the following six books. To order individual copies, use the following order numbers.	SBOF-1627	
Communications Guide	SC34-0638	
Extended Address Mode and Performance Analyzer User Guide	SC34-0591	
Installation and System Generation Guide	SC34-0646	
Language Reference	SC34-0643	
Library Guide and Common Index	SC34-0645	
Messages and Codes	SC34-0636	
Operator Commands and Utilities Reference	SC34-0644	
Guides and reference cards:		
Set of the following four books and reference cards. To order individual copies, use the following order numbers.	SBOF-1628	
Customization Guide	SC34-0635	
Event Driven Language Programming Guide	SC34-0637	
Operation Guide	SC34-0642	
Problem Determination Guide	SC34-0639	
Language Reference Card	SX34-0165	
Operator Commands and Utilities Reference Card	SX34-0164	
Conversion Charts Reference Card	SX34-0163	
Reference Card Envelope	SX34-0166	
Set of three reference cards and storage envelope. (One set is included with order number SBOF-1627.)	SBOF-1629	
Binders:		
3-ring easel binder with 1 inch rings	SR30-0324	
3-ring easel binder with 2 inch rings	SR30-0327	
Standard 3-ring binder with 1 inch rings	SR30-0329	
Standard 3-ring binder with 1 1/2 inch rings	SR30-0330	
Standard 3-ring binder with 2 inch rings	SR30-0331	
Diskette binder (Holds eight 8-inch diskettes.)	SB30-0479	

Publications Order Form

Cut or Fold 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, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM Corporation
1 Culver Road
Dayton, New Jersey 08810

Fold and tape — Please Do Not Staple — Fold and tape

IBM®

International Business Machines Corporation

IBM Series/1 Event Driven Executive
Language Reference

Order No. SC34-0643-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

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 or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC34-0643-0

Printed in U.S.A.

Reader's Comment Form

Cut or Fold 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, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Information Development, Department 28B
P.O. Box 1328
Boca Raton, Florida 33432

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

IBM®

IBM Series/1 Event Driven Executive
Language Reference

Order No. SC34-0643-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.* Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

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 or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC34-0643-0
Printed in U.S.A.

Reader's Comment Form

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, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Information Development, Department 28B
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape   Please Do Not Staple   Fold and tape

IBM ®

Cut or Fold Along Line

IBM Technical Newsletter

This Newsletter No.	SN34-0938
Date	4 June 86
Base Publication No.	SC34-0643-1
File No.	S1-35
Previous Newsletters	None

IBM Series/1
Event Driven Executive
Language Reference
Program Numbers: 5719-XS5, 5719-AM4, 5719-CX1, 5719-MS2, 5719-SX1, 5719-XX9

©IBM Corp. 1984, 1985, 1986

This Technical Newsletter, a part of Version 5 Modification Level 2 of the Event Driven Executive, provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent levels unless specifically altered. Pages to be inserted and/or removed are:

iii, iv
LR-127, LR-128
LR-239, LR-240
LR-488.1, LR-488.2 (added)
LR-493, LR-494
LR-523 through LR-528
LR-613 through LR-622

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

Summary of Amendments

This Technical Newsletter contains the following additions or modifications to text:

- The TERMCTRL section has been updated with information on the 4201 printer's operation after you power it off and then on again.
- The TERMCTRL section has been updated with information on the 4224 printer's operation after you power it off and then on again.
- The $IMAGE subroutines in Appendix A, "Formatted Screen Subroutines", have been updated with information for coding static screen images on the 3161, 3163, and 3164 display terminals.
- Miscellaneous editorial updates.

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

IBM Corporation, Information Development (3405), Department 28B, P.O. Box 1328, Boca Raton, Florida 33429-1328

Printed in U.S.A.

Summary of Changes For Version 5.2

This document contains the following changes.

- "READTEXT - Read text entered at a terminal" on page LR-401 has been updated to include information about 3161, 3163, and 3164 terminals operating in block mode.
- One new and several updated SNA instructions, their syntax and descriptions appear in Chapter 2, "Instruction and Statement Descriptions" beginning on page LR-299.
- Information on coding TERMCTRL instructions for terminal models 3161, 3163, and 3164 appears in this edition. Refer to "3101, 3161, 3163, and 3164 Display Terminals (Block Mode)" on page LR-470 for details.
- Information on coding TERMCTRL instructions for the 4224 and 4201 Printers also appears in this edition. Details are located under "4201 Printer" on page LR-475 and"4224 Printer" on page LR-489.
- Information on the 4201 and 4224 printer operations after you power them off and then on again has been included in this edition. Refer to "Special Considerations" on page LR-522 and"Special Considerations" on page LR-488 respectively for details.
- The $IMAGE section has been updated with information on coding static screen images for the terminal models 3161, 3163, and 3164. Refer to Appendix A, "Formatted Screen Subroutines" on page LR-613 for information.

Summary of Changes For Version 5.2

- A sample Tape Source Dump Utility program has been added to Appendix D. Refer to "Tape Source Dump Program Example" on page LR-676 for information.
- The "Glossary of Terms and Abbreviations" for this document is now located in the *Library Guide and Common Index*.

A vertical line in the left margin indicates new or changed material.

DO - Perform a program loop

The DO instruction begins a program loop. A loop is a set of one or more instructions that executes repeatedly until a condition you specify in the DO instruction is satisfied. You must end the DO loop with an ENDDO instruction.

You can code a loop within another loop. This technique is called "nesting." You can include up to 20 nested loops within your initial DO-ENDDO structure.

There are three forms of the DO instruction. DO UNTIL and DO WHILE provide a means of looping until or while a condition 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.

You also can use the DO instruction to perform a loop while or until a certain bit is 'on' (set to 1) or 'off' (set to 0).

The syntax box shows the DO UNTIL and DO WHILE forms of the DO instruction with a single conditional statement. You can specify several conditional statements, however, by using the AND and OR keywords. These keywords allow you to join conditional statements. The keywords are described in the operands list and examples using the keywords are shown under "Syntax Examples with DO and ENDDO" on page LR-130.

Syntax:

```
label        DO       count,TIMES,INDEX=,P1=
label        DO       UNTIL,(data1,condition,data2,width)
label        DO       WHILE,(data1,condition,data2,width)

Required:    count or one conditional statement
             with UNTIL or WHILE
Defaults:    width is WORD
Indexable:   count or data1 and data2 in each statement
```

Operand	***Description***
count	The number of times the loop is to be executed. You can specify a constant or the label of a variable. The maximum value is 32767. The system completes one loop each time it encounters the ENDDO instruction. **Note:** If count=0, the system executes the loop one time.
TIMES	This optional operand serves only as a comment for the count operand.
INDEX=	The label of a data area that the system resets to 0 before starting the DO loop and increases by 1 each time the instruction following the DO instruction executes. The first time the DO loop executes, the index has a value of 1.

DO - Perform a program loop *(continued)*

UNTIL This operand defines a loop that executes until the condition you specify is true. The loop executes at least once, even if the condition is initially true.

WHILE This operand defines a loop that executes as long as the condition you specify is true. The loop does not execute if the condition is initially false.

data1 The label of a data item to be compared to data2 or the label of the data area that contains the bit to be tested. This operand is valid only in a conditional statement with UNTIL or WHILE.

condition An operator that indicates the relationship or condition to be tested. Only code this operand in a conditional statement with UNTIL or WHILE. The valid operators for the DO instruction are as follows:

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

ON - Bit is 'on'
OFF - Bit is 'off'

data2 The data to be compared to data1 or the position, in data1, of the bit to be tested. Only code this operand in a conditional statement with UNTIL or WHILE. You can specify immediate data or the label of a variable. Immediate data can be an integer between 1 and 32768 or a hexadecimal value between 1 and 65535 (X'FFFF').

Bit 0 is the left-most bit of the data area.

width Specifies an integer number of bytes or one of the following:

BYTE	- Byte (8 bits)
WORD	- Word (16 bits)
DWORD	- Doubleword (32 bits)
FLOAT	- Single-precision floating-point (32 bits)
DFLOAT	- Extended-precision floating-point (64 bits)

Code this operand only in a conditional statement using UNTIL or WHILE. The default is WORD.

AND Enables you to join conditional statements when you code DO UNTIL or DO WHILE. Code the operand between the conditional statements you want to join. With DO UNTIL, the AND indicates that the loop should execute *until* all the conditional statements that the operand joins are true. With DO WHILE,

IF - Test if a condition is true or false

The IF instruction determines whether a conditional statement is true or false and, based on its decision, determines the next instruction to execute.

A conditional statement can compare two data items or ask whether a bit is "on" (set to 1) or "off" (set to 0). The instruction syntax shows the general format of conditional statements used with the IF instruction.

You can compare data in two ways: *arithmetically* or *logically*. When you compare data arithmetically, the system interprets each number as a positive or negative value. The system, for example, interprets X'0FFF' as 4095. It interprets X'FFFF', however, as a -1. Though X'FFFF' seems to be a larger hexadecimal number than X'0FFF', the system recognizes the former as a negative number and the latter as a positive number. X'FFFF' is a negative number to the system because the left-most bit is "on."

When you compare data logically, the system compares the data areas byte by byte. The system interprets X'FFFF' not as a -1 but as a string of 2 bytes with all bits "on."

With EBCDIC or ASCII character data, the system makes a logical comparison of the characters byte by byte. In a logical comparison of a capital 'A' (X'C1') with a capital 'H' (X'C8'), the system recognizes the capital A to be "less than" the capital H. By comparing character data logically, you can use the IF instruction to sort items alphabetically ('a' is less than 'c' which is greater than 'b').

The syntax box shows the IF instruction with a single conditional statement. You can specify several conditional statements on a single IF instruction, however, by using the AND and OR keywords. These keywords allow you to join conditional statements. "Rules for Evaluating Statement Strings Using AND and OR" on page LR-129 provides additional information regarding use of the IF instruction. The keywords are described in the operands list and examples using the keywords are shown following the instruction description.

Syntax:

```
label       IF          (data1,condition,data2,width)

label       IF          (data1,condition,data2,width),GOTO,loc

Required:   one conditional statement
Defaults:   width is WORD for arithmetic comparison
Indexable:  data1 and data2 in each statement
```

IF - Test if a condition is true or false *(continued)*

Operand	*Description*
data1	The label of a data item to be compared to data2 or the label of the data area that contains the bit to be tested.
condition	An operator that indicates the relationship or condition to be tested. The valid operators for the IF instruction are as follows:

Arithmetic and Logical Comparisons	*Testing a Bit Setting*
EQ - Equal to	ON or OFF
NE - Not equal to	
GT - Greater than	
LT - Less than	
GE - Greater than or equal to	
LE - Less than or equal to	

data2 The label of a data item to be compared to data1 or the label of the data area that contains the bit in data1 to be tested. For an arithmetic comparison, specify immediate data or the label of a data area. Immediate data can be an integer from 0 to 32767, or a hexadecimal value from 0 to 65535 (X'FFFF'). For a logical comparison, specify the label of a data area. For a bit comparison, specify immediate data.

When you check a bit setting, remember that bit 0 is the leftmost bit of the data area.

width Specify an integer number of bytes in the range of 1 to 65535 for a logical comparison (no default). For a bit comparison, specify an immediate data area in words. This form specifies that both DATA1 and DATA2 are storage locations; an immediate operand is not permitted.

For an arithmetic comparison, you can specify one of the following:

BYTE	- Byte (8 bits)
WORD	- Word (16 bits), the default
DWORD	- Doubleword (32 bits)
FLOAT	- Single-precision floating-point (32 bits)
DFLOAT	- Extended-precision floating-point (64 bits)

GOTO If the statement is true and GOTO is coded, control passes to the instruction at the address specified in the loc operand. If the statement is false, execution proceeds sequentially.

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

TERMCTRL - Request special terminal function *(continued)*

- If you power off and then power on the 4201, the printer resets the following functions as shown:

Function	Hardware Default
BOLD	Off
DSTRIKE	Off
DWIDE	Off
LPI	6 LPI
OVER	Off
PDEN	10 CPI
SETFONT	Data Processing
SUBSCRIPT	Off
SUPERSCRIPT	Off
UNDER	Off

This page intentionally left blank.

TERMCTRL - Request special terminal function *(continued)*

Syntax:

```
label        TERMCTRL  BARCODE,loc,count,XCOORD=,YCOORD=,
                       ORIENT=,BARTYPE=,MOD=,HEIGHT=,WIDTH=,.
                       P2=,P3=,P4=,P5=,P6=,P7=,P8=,P9=,P10=

Required:    BARCODE,loc,count,XCOORD=,YCOORD=,MOD=
Defaults:    ORIENT=HORZ,BARTYPE=CODE3#9,HEIGHT=0
             WIDTH=NARROW
Indexable:   loc,count
```

Operand	*Description*
BARCODE	Causes the 4224 to print a bar code. The printer defers the actual printing of the bar code until other data being printed causes the print head to reach the specified "X" and "Y" coordinates. Issue the BARCODE command at the top of a page to be sure the printer receives it before the print head reaches the point where the bar code is to be placed.

If the bar code is sent to the printer after the print head has passed the starting point of the desired print location, the 4224 may try to print what it can of the bar code and will generate a hardware error indicating an invalid request for backward movement of the print head. For this reason, applications must issue a BARCODE command before the print head reaches the point on the physical page where the bar code is to begin.

Since bar code printing is completely independent of immediate (normal) printing, the application must anticipate where the bar code will be placed and skip the appropriate number of spaces and lines to avoid overwriting. Select the location of a bar code on a page with the XCOORD= and YCOORD= operands.

The 4224 prints bar codes in black only, regardless of the currently active color.

ORIENT= Orientation of the bar code. Allowable values are:

Parameter	*Description*
HORZ	Orient the bar code horizontally, the default.
VERT	Orient the bar code vertically.

TERMCTRL (4224)

TERMCTRL - Request special terminal function *(continued)*

loc The label of characters the system will encode and print in the bar code you selected. The system *does not* translate this data before sending it to the printer.

count Count of characters the system will encode and print in the bar code you selected. Valid counts are listed below for each bar code type under BARTYPE=.

XCOORD= Word value in 1/1440 inch units specifying the location on the current page where the bar code will be printed (upper left corner of the bar code). The printer resolves the coordinates to the nearest increment it supports (1/144 inch). Specify the X coordinate relative to the left edge of the physical page.

YCOORD= Word value in 1/1440 inch units specifying the location on the current page where the bar code will be printed (upper left corner of the bar code). The printer resolves the coordinates to the nearest increment it supports (1/144 inch). Specify the Y coordinate relative to the top of the page.

BARTYPE= Word value specifying the type of bar code desired.

Mnemonic	Count (Bytes)	Bar Code Description
CODE3#9	1–50	Code 3 of 9
MSI	1–50	MSI (MSI Data Corporation)
UPC#A	11	Uniform Product Code - Type A
UPC#E	10	Uniform Product Code - Type E
UPC#2	2	UPC - Magazine and Paperback - two digit
UPC#5	5	UPC - Magazine and Paperback - five digit
EAN#8	7	European Article Number - Type 8
EAN#13	12	European Article Number - Type 13
INDUST	1–50	Two of Five Industrial
MATRIX	1–50	Two of Five Matrix
LEAVED	1–50	Two of Five Interleaved.

Note: You may select supplemental encoding EAN#2 and EAN#5 by specifying bartypes UPC#2 and UPC#5 respectively.

TERMCTRL - Request special terminal function *(continued)*

Applications that currently run on the 4975-02L printer will run on the 4224 printer without reassembly with the exceptions noted in this section. However, a new system generation is required and applications must be relinked to include the modified $4975 module.

To take advantage of any new function provided by the 4224 printer, you must modify and reassemble your 4975-02L printer applications. If you decide to modify your application, you can avoid relinking with module $4975 by replacing the TERMCTRL SET instructions in your program with corresponding TERMCTRL instructions for the 4224 printer as follows:

4975-02L Instruction	4224 Instruction
SET,LPI=	LPI,HEIGHT=
SET,PMODE=	SETFONT,FONTID=
SET,PDEN=	PDEN,DENSITY=
SET,CHARSET=	(OFFLINE TEST 303)
SET,RESTORE	RESTORE

If you decide not to reassemble your application, note the following:

- PMODE=TEXT on the 4224 printer produces near letter quality, proportionally-spaced characters with a single pass of the print head (FONTID=5). PMODE=TEXT directs the 4224 to reset the print density to large and to redefine horizontal densities. See TERMCTRL SET for more information. PMODE=TEXT on the 4975-02L printer produces TEXT quality, proportionally-spaced characters with two passes of the print head. PMODE=TEXT directs the 4975-02L to select the appropriate density for the proportionally-spaced characters.

- PMODE=TEXT1 on both the 4975-02L and the 4224 printer produces TEXT quality proportionally-spaced characters with a single pass of the print head (FONTID=4 on the 4224). PMODE=TEXT1 directs the 4975-02L to select the appropriate density for the proportionally-spaced characters. PMODE=TEXT1 directs the 4224 to reset the print density to large and to redefine horizontal densities. See TERMCTRL SET for more information.

- PMODE=DRAFT on both the 4975-02L and the 4224 produces data processing quality, monospaced characters with a single pass of the print head.

  Note: Near letter quality is a higher quality type than text quality.

- The TERMCTRL DCB= operand of the 4975-02L is not supported on the 4224 printer.

- TERMCTRL SET,CHARSET= is a null operation on the 4224 printer. You may select a character set for languages other than English by running offline test 303. Refer to TERMCTRL SET,CHARSET= for additional information.

- TERMCTRL SET,PMODE=TEXT or TEXT1 on the 4975-02L printer produces approximately 5 CPI. TERMCTRL SET,PMODE=TEXT or TEXT1 on the 4224 printer, however, produces approximately 8, 10 or, 12 CPI (depending on the density selected).

TERMCTRL (4224)

TERMCTRL - Request special terminal function *(continued)*

To produce approximately 5 CPI on the 4224 printer, simulating the 4975-02L, issue TERMCTRL DWIDE and TERMCTRL PDEN,DENSITY=NORMAL after issuing TERMCTRL SET,PMODE=TEXT or TEXT1.

- TERMCTRL SET,PDEN= values (print densities in characters per inch) for the 4975-02L and 4224 printers differ in the following manner:

Density	4975-02L Printer	4224 Printer
Compressed	COMP=20	COMP=15
Normal	NORM=15	NORM=15
Expanded	EXPD=10	EXPD=10

- If you power off and then power on the 4224, the printer resets the following functions as follows:

Function	Hardware Default
BARCODE	Deleted (if pending)
BOLD	Off
CHARSET	Offline test 303 value
DSTRIKE	Off
DWIDE	Off
ITALICS	Off
Loaded fonts	Deleted
LPI	Offline test 302 value
OVER	Off
PCOLOR	Black
PDEN	Offline test 302 value
SETFONT	Offline test 302 value
SUBSCRIPT	Off
SUPERSCRIPT	Off
UNDER	Off

- Data streaming mode is supported to allow the user access to features of the 4224 printer not implemented. Issuing a PRINTEXT with XLATE=NO activates data streaming mode.

  Text data to be sent to the 4224 printer is not translated when XLATE=NO is coded. Each PRINTEXT, XLATE=NO is counted by the printer support as a single line even though multiple physical lines may be printed. Therefore, when switching from untranslated mode to translated mode, you may want to issue a PRINTEXT LINE=0 before issuing translated commands in order to synchronize the hardware and the software. For details on the printer data stream, refer to the IBM *4224 Printer Product and Programming Description Manual*, GC31-2550.

TERMCTRL - Request special terminal function *(continued)*

Additional 4224 Printer Information

- The 4224 printer can only be attached locally. Remote attachment of the 4224 printer, unlike the 4975-02L, is not possible.

- Not all $TERMUT1 and $TERMUT2 utility functions of the 4975-02L printer are directly available on the 4224 printer. Refer to information on the use of these utilities with the 4224 and 4975-02L printers in the *Operator Commands and Utilities Reference*.

- The 4224 printer maintains physical page size in inches. You select the initial physical page size using offline test 302. The 4224 printer maintains logical page size as a line count. Whenever you change logical page size with ENQT, DEQT, or $TERMUT1, be sure to alter line height so that: (physical page size in inches) x (lines per inch) = (logical page size).

- The 4224 printer supports both ASCII and EBCDIC character sets.

  The different models of the 4224 are indistinguishable to the EDX printer support. Variations among the printer models follow:

  - Model 301 — runs at 200 characters per second (top speed). It has only one color (black).

  - Model 302 — runs at 400 characters per second (top speed). It has only one color (black).

  - Model 3C2 — runs at 400 characters per second (top speed). It supports up to eight colors depending on which ribbon is installed.

- If the green light on the 4224 flashes after you have cancelled your application, you may empty the printer's buffer as follows:

  1. Press the "STOP" button on the 4224 printer.
  2. Press the "ALT" and "CANCEL" buttons to clear the 4224 print buffer.
  3. Press the "START" button on the 4224 printer.

- PRINTEXT instructions issued to the 4224 printer return the ACCA return codes listed under "PRINTEXT - Display a message on a terminal" on page LR-339.

- To interpret the ISB after an I/O completion error, refer to the hardware manual of the Series/1 attachment being used to drive the 4224 printer (MFA or 2095/2096). To interpret the ISB after an error is reported as an attention interrupt, refer to the IBM *4224 Printer Product and Programming Description Manual*, GC31-2550.

TERMCTRL (4224)

TERMCTRL - Request special terminal function *(continued)*

- If you have issued an ENQT with an IOCB and provided a local buffer to be used instead of the terminal control block (CCB) buffer, remember the following.
  - Do not alter the buffer in any way (except for direct I/O) during the time when the buffer is in use as a system buffer.
  - The printer support issues additional I/O operations because the same buffer must be used for both application data and TERMCTRL data. This degrades performance.
  - The right margin on the 4224 printer is automatically set to buffer size + left margin -1, regardless of the value you specify for RIGHTM=. If you exceed the physical right margin of the 4224, the extra data is printed on the next line.

Programming Aids

All mnemonics have associated equates that can be used to generate values during execution. The equate is the same as the mnemonic, but it has a # in front of it. You can find the equates in the copy code module EQU4224.

The bar code orientation mnemonics have the following equates:

Mnemonic	Equate	Equate Value
HORZ	#HORZ	0
VERT	#VERT	1

The BARTYPE= mnemonics have the following equates:

Mnemonic	Equate	Equate Value
CODE3#9	#CODE3#9	1
MSI	#MSI	2
UPC#A	#UPC#A	3
UPC#E	#UPC#E	5
UPC#2	#UPC#2	6
UPC#5	#UPC#5	7
EAN#8	#EAN#8	8
EAN#13	#EAN#13	9
INDUST	#INDUST	10
MATRIX	#MATRIX	11
LEAVED	#LEAVED	12

The WIDTH= mnemonics have the following equates:

Mnemonic	Equate	Equate Value
NARROW	#NARROW	14
WIDE	#WIDE	21

TERMCTRL - Request special terminal function *(continued)*

The CHARID= mnemonics have the following equates:

Mnemonic	Equate	Equate Value
KANA	#KANA	0
PC1	#PC1	1
PC2	#PC2	2
INT1	#INT1	3
INT5	#INT5	4
APL	#APL	5

The DENSITY= mnemonics have the following equates:

Mnemonic	Equate	Equate Value
LARGE	#LARGE	0
NORMAL	#NORMAL	1
DENSE	#DENSE	2

The PCOLOR= mnemonics have the following equates:

Mnemonic	Equate	Equate Value
BLUE	#BLUE	1
RED	#RED	2
MAGENTA	#MAGENTA	3
GREEN	#GREEN	4
CYAN	#CYAN	5
YELLOW	#YELLOW	6
BLACK	#BLACK	8
BROWN	#BROWN	16

Equate values should never be hard-coded. Either the mnemonic should be used (when the value is known at assembly time), or the equate should be used (for run time recognition).

Coding Example

Examples of accessing a color at run time are:

```
          MOVE         #1,+#BLUE         USE COLOR BLUE
          TERMCTRL     PCOLOR,COLOR=#1   SET DESIRED COLOR
          .
          .
          .
          TERMCTRL     PCOLOR,COLOR=SKYBLUE   SET COLOR BLUE
          .
          .
          .
          MOVEA        #1,SKYBLUE        POINT TO BLUE
          TERMCTRL     PCOLOR,COLOR=(0,#1)    SET DESIRED COLOR
          .
          .
          .
SKYBLUE   DATA         A(+#BLUE)         COLOR BLUE
```

All equates for the 4224 printer are word values. Be sure to define them as such in storage with the data definition A(+equate).

TERMCTRL - Request special terminal function *(continued)*

4973 Printer

Syntax:

```
label         TERMCTRL  function,LPI=,DCB=

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

Operand	*Description*	
function:		
	SET	Sets the number of lines per inch and causes any buffered output to be printed. The system also resets the current output position to the beginning of the left margin. When you specify SET, you must also specify LPI.
	DISPLAY	Causes the system to write to the 4973 any buffered output.

LPI= The number of lines per inch (either 6 or 8) the 4973 is to print. This operand is required when the SET function is specified.

DCB= The label of an 8-word device control block you define with the DCB statement. The 4973 support code provides an IDCB that points to this DCB and issues a START I/O instruction to the device. The system does a wait operation and control returns to you after the interrupt is received from the device.

If the post-cursor bit is set on in word 0 of the DCB, the terminal support updates the internal cursor position according to word 1 of the DCB. If an error occurs, an error return is made according to normal terminal I/O conventions.

Do not code any other operands when you specify this operand on the TERMCTRL statement. You cannot have another DCB chained to the one specified by the DCB operand. You should be familiar with the 4973 hardware and terminal I/O internals when you use this operand.

Appendix A. Formatted Screen Subroutines

You can create, save, and modify formatted screen images using the $IMAGE utility. Refer to the $IMAGE description in the *Operator Commands and Utilities Reference* for information on creating or exchanging terminal screen images for various terminals. The formatted screen subroutines retrieve and display these images. This appendix describes each of the following subroutines and its operands:

- $IMDATA
- $IMDEFN
- $IMOPEN
- $IMPROT
- $PACK
- $UNPACK.

You can use the formatted screen subroutines with the following terminals:

- 4978 terminals
- 4979 terminals
- 4980 terminals
- 3101 terminals in block mode
- 3161 terminals in block mode
- 3163 terminals in block mode
- 3164 terminals in block mode.

You can also use screen images created on a 4978, 4979, or 4980 on any of the terminals listed above by calling subroutines described in this appendix.

Formatted Screen Subroutines

You must code an EXTRN statement for each subroutine name to which your program refers. You also must link-edit the subroutines with your application program. Specify $AUTO,ASMLIB as the autocall library to include the screen formatting subroutines. Refer to the *Operator Commands and Utilities Reference* for details on the AUTOCALL option of $EDXLINK.

You call the formatted screen subroutines using the CALL instruction. The following section shows the CALL instruction syntax for each subroutine.

If an error occurs, the terminal I/O return code is in the first word of the task control block (TCB). These errors can come from instructions such as PRINTEXT, READTEXT, and TERMCTRL.

$IMDATA Subroutine

The $IMDATA subroutine displays the initial data values for an image which is in disk storage format. Use $IMDATA:

- To display the unprotected data associated with a screen image, if the buffer contains a screen format retrieved with $IMOPEN.
- To "scatter write" the contents of a user buffer to the input fields of a displayed screen image.

Note: You must call $IMDATA if any of your unprotected fields have the right justify or must enter characteristics.

If the buffer is retrieved with $IMOPEN, the buffer begins with the characters "IMAG," or "IM31," and the buffer index (buffer-4) equals the data length excluding the characters "IMxx."

You can specify a user buffer containing application-generated data. Set the first four bytes of the buffer to the characters "USER" and set the buffer index (buffer-4) to the data length excluding the characters USER.

All or portions of the screen may be protected after $IMDATA executes. Because the operator cannot key data into protected fields, subsequent read instructions (such as QUESTION, GETVALUE, and READTEXT) should be directed to unprotected areas of the screen, or the protected areas should be erased.

Notes:

1. To use $IMDATA, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.
2. Do not call both $IMDATA and $IMPROT by separate tasks to operate simultaneously. Problems will occur because both call the $IMDTYPE subroutine.

Syntax:

```
label        CALL        $IMDATA,(buffer),(ftab),P2=,P3=

Required:    buffer,ftab  (see note)
Defaults:    none
Indexable:   none
```

$IMDATA

$IMDATA Subroutine *(continued)*

Operand	*Description*
buffer	The label of an area containing the image in disk-storage format.
ftab	The label of a field table constructed by $IMPROT giving the location (lines,spaces) and size (characters) of each unprotected data field of the image. **Note:** The ftab operand is required only if the application executes on a 3101, 3161, 3163, or 3164 terminal in block mode, or if a user buffer is used in $IMDATA.
Px=	Parameter naming operands. See "Using the Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMDATA Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
9	Invalid format in buffer
12	Invalid terminal type

$IMDEFN Subroutine

The $IMDEFN subroutine creates an IOCB for the formatted screen image. You can code the IOCB directly, but the use of $IMDEFN allows the image dimensions to be modified with the $IMAGE utility without requiring a change to the application program. $IMDEFN updates the IOCB to reflect OVFLINE=YES. Refer to the TERMINAL configuration statement in the *Installation and System Generation Guide* for a description of the OVFLINE parameter.

Once you define an IOCB for the static screen, the program can then acquire that screen through ENQT. Once the screen has been acquired, the program can call the $IMPROT subroutine to display the image and the $IMDATA subroutine to display the initial nonprotected fields.

Note: To use $IMDEFN, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

```
label        CALL         $IMDEFN,(iocb),(buffer),topm,leftm,
                          P2=,P3=,P4=,P5=

Required:    iocb,buffer
Defaults:    topm=0,leftm=0
Indexable:   none
```

Operand	***Description***
iocb	The label of an IOCB statement defining a static screen. The IOCB need not specify the TOPM, BOTM, LEFTM, nor RIGHTM parameters; these are "filled in" by the subroutine. The following IOCB statement would normally suffice: `label IOCB SCREEN=STATIC`
buffer	The label of an area containing the screen image in disk storage format. The format is described in the *Event Driven Executive Language Programming Guide.*
topm	This parameter indicates the screen position at which line 0 will appear. If its value is such that lines would be lost at the bottom of the screen, then it is forced to zero. This parameter must equal zero for all 3101, 3161, 3163, or 3164 terminal applications. The default is also zero.
leftm	This parameter indicates the screen position at which the left edge of the image will appear. If its value is such that characters would be lost at the right edge of the screen, then it is forced to zero. This parameter must equal zero for all 3101, 3161, 3163, or 3164 terminal applications. The default is also zero.
Px=	Parameter naming operands. See "Using the Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMDEFN Subroutine *(continued)*

Coding Example

```
         CALL     $IMDEFN,(IMGIOCB),(IMGBUFF),0,0
           .
           .
           .
         ENQT     IMGIOCB
           .
           .
           .
         PROGSTOP
IMGIOCB  IOCB     SCREEN=STATIC
IMGBUFF  BUFFER   1024,BYTES
```

$IMOPEN Subroutine

The $IMOPEN subroutine reads a formatted screen image from disk or diskette into your program buffer. You can also perform this operation by using the DSOPEN subroutine or by defining the data set at program load time, and issuing the disk READ instruction. Refer to the *Event Driven Executive Language Programming Guide* for a description of buffer sizes. $IMOPEN updates the index word of the buffer with the number of actual bytes read. To refer to the index word, code buffer-4.

Note: To use $IMOPEN, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

Syntax:

```
label        CALL        $IMOPEN,(dsname),(buffer),(type),
                         P2=,P3=,P4=

Required:    dsname,buffer
Defaults:    type=C'4978'
Indexable:   none
```

Operand	*Description*
dsname	The label of a TEXT statement which contains the name of the screen image data set. You can include a volume label, separated from the data set name by a comma.
buffer	The label of a BUFFER statement that defines the storage area into which the image data will be read. Allocate the storage in bytes, as in the following example:

```
label      BUFFER   1024,BYTES
```

type The label of a DATA statement that reserves a 4-byte area of storage and specifies the type of image data set to be read. The DATA statement must be on a full word boundary. Specify one of the following types:

C'4978'	The system reads an image data set for a 4978 terminal with a 4978/4979/4980 terminal format. This is the default terminal format.
C'3101'	The system reads an image data set for a 3101 terminal with a 31xx terminal format.
C'3161'	The system reads an image data set for a 3161 terminal with a 31xx terminal format.

$IMOPEN

$IMOPEN Subroutine *(continued)*

C'3163' The system reads an image data set for a 3163 terminal with a 31xx terminal format.

C'3164' The system reads an image data set for a 3164 terminal with a 31xx terminal format.

Note: The 31xx terminal format is the format used for a 3101, 3161, 3163, and 3164 terminal.

C' ' The system reads an image data set whose format corresponds with the type of terminal enqueued. If neither a 4978, 4979, 4980, 3101, 3161, 3163, nor 3164 is enqueued (ENQT), the system assumes the default 4978 image format.

If you use this option, $IMOPEN will try to use the format that corresponds with the device. If that is not available, $IMOPEN will use a 4978/4979/4980 screen image. This is the default condition when you do not code this parameter. For example, if you are enqueued on a 3161 terminal, $IMOPEN will attempt to open a 31xx screen image. If it does not exist, it will use the 4978 screen image.

Px= Parameter naming operands. See "Using the Parameter Naming Operands (Px=)" on page LR-12 for a description of these operands.

$IMOPEN Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
1	Disk I/O error
2	Buffer too small for 3101, 3161, 3163, or 3164 terminal information (31xx screen image)
3	Data set not found
4	Incorrect header or data set length
5	Input buffer too small
6	Invalid volume name
7	No 3101 image available
8	Data set name longer than eight bytes

$IMPROT Subroutine

The $IMPROT subroutine uses an image created by the $IMAGE utility to prepare the defined protected and blank nonprotected fields for display. At the option of the calling program, a field table can be constructed. The field table gives the location (LINE and SPACES) and length of each unprotected field.

Upon return from $IMPROT, your program can force the protected fields to be displayed by issuing a TERMCTRL DISPLAY. This is not required if a call to $IMDATA follows because $IMDATA forces the display of screen data.

All or portions of the screen may be protected after $IMPROT executes. Because the operator cannot key data into protected fields, subsequent read instructions (such as QUESTION, GETVALUE, and READTEXT) should be directed to unprotected areas of the screen, or the protected areas should be erased.

Notes:

1. To use $IMPROT, you must code an EXTRN statement in your program. You must also link-edit the program with $EDXLINK and specify an autocall to $AUTO,ASMLIB.

2. Do not call both $IMPROT and $IMDATA by separate tasks to operate simultaneously. Problems will occur because both call the $IMDTYPE subroutine.

Syntax:

```
label        CALL        $IMPROT,(buffer),(ftab),P2=,P3=

Required:    buffer,ftab  (see note)
Defaults:    none
Indexable:   none
```

Operand	***Description***
buffer	The label of an area containing the screen image in disk storage format. The format is described in the *Event Driven Executive Language Programming Guide*.
ftab	The label of a field table constructed by $IMPROT giving the location (lines, spaces) and size (characters) of each unprotected data field of the image. **Note:** The ftab operand is required only if the application executes on a 3101, 3161, 3163, or 3164 terminal in block mode, or if a user buffer is used in $IMDATA.
Px=	Parameter naming operands. See "Using the Parameter Naming Operands (Px=)" on page LR-12 for a description of how to use these operands.

$IMPROT Subroutine *(continued)*

The field table has the following form:

```
label-4         number of fields
label-2         number of words
label           line      * FIELD 1    (one word)
                spaces                 (one word)
                size                   (one word)
label+6         line      * FIELD 2
                spaces
                size
                .
                .
                .
label+6(n-1)    line      * FIELD n
                spaces
                size
```

The field numbers correspond to the following ordering: left to right in the top line, left to right in the second line, and so on to the last field in the last line. Storage for the field table should be allocated with a BUFFER statement specifying the desired number of words using the WORDS parameter. The buffer control word at label-2 is used to limit the amount of field information stored, and the buffer index word at buffer-4 is set with the number of fields for which information was stored, the total number of words being three times that value. If the field table is not desired, code zero for this parameter.

$IMPROT Return Codes

The return codes are returned in the second word of the task control block (TCB) of the program or task calling the subroutine. The label of the TCB is the label of your program or task (taskname). Refer to taskname+2.

Code	Description
-1	Successful completion
9	Invalid format in buffer
10	Ftab truncated due to insufficient buffer size
11	Error in building ftab from 31xx terminal format; partial ftab created
12	Invalid terminal type

IBM®

International Business Machines Corporation

SC34-0643-0

SC34-0643-0
Program Numbers: 5719-SX4, 5719-AM4,
5719-CX1, 5719-MS2, 5719-SX1
File Number: S1-35
Printed in U.S.A.