IBM

Series/1

SC34-0438-0

Event Driven Executive Language Programming Guide

Version 4.0

Library Guide and Common Index	Installation and System Generation Guide	Operator Commands and Utilities Reference
Language Reference	Communications Guide	Messages and Codes
Operation Guide	Event Driven Language Programming Guide	Reference Cards
Problem Determination Guide	Customization Guide	Internal Design

IBM

Series/1

SC34-0438-0

Event Driven Executive Language Programming Guide

Version 4.0

Library Guide and Common Index	Installation and System Generation Guide	Operator Commands and Utilities Reference
Language Reference	Communications Guide	Messages and Codes
Operation Guide	Event Driven Language Programming Guide	Reference Cards
Problem Determination Guide	Customization Guide	Internal Design

First Edition (May 1983)

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

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

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, Department 27T, 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.

About This Book

This book contains an introduction to the Event Driven Language.

Audience

Chapters 1 through 8 of this book are intended for the application programmer who is coding in the Event Driven Language for the first time. Readers should be familiar with basic data processing terminology and concepts, such as input, output, and data sets.

Chapters 9 through 18 are intended for application programmers who need information about such advanced topics as multitasking, data management from a program, communicating with other programs, and writing graphics or sensor I/O programs.

How This Book is Organized

This book contains eighteen chapters and three appendixes:

- *Chapter 1. Getting Started* describes the steps necessary to develop and run a simple Event Driven Language (EDL) program.
- *Chapter 2. Writing a Source Program* tells how to use EDL instructions to do such things as read data, write data, convert data, and manipulate data.

How This Book is Organized *(continued)*

- *Chapter 3. Entering a Source Program* tells how to use the full-screen editor to enter and modify a source program.
- *Chapter 4. Compiling a Source Program* shows how to use the Event Driven Language compiler to translate a source program to object code.
- *Chapter 5. Preparing Object Code for Execution* shows how to use the linkage editor to prepare an object program for execution.
- *Chapter 6. Executing a Program* describes how to run a program that has been compiled and link-edited.
- *Chapter 7. Finding and Fixing Errors* describes a tool you can use to diagnose program logic errors and exception conditions.
- *Chapter 8. Reading and Writing Data from Screens* shows how to read and write data from display terminals. The chapter defines roll screens and static screens and describes how to write programs that interact with the operator.
- *Chapter 9. Designing Complex Programs* defines what a program and a task are and describes multitasking, subroutines, program overlays, segment overlays, and unmapped storage.
- *Chapter 10. Performing Data Management from an Application Program* describes various ways to do data management from a program. The chapter describes how to allocate, delete, rename, and open a data set. In addition, the chapter shows how to set the logical end of file, add records to a tape data set, and find device type from a program.
- *Chapter 11. Coding Programs That Use Tape* tells how to read to and write from a magnetic tape data set.
- *Chapter 12. Communicating with Another Program (Cross Partition Services)* shows how programs can interact with each other, either within the same partition or between partitions.
- *Chapter 13. Communicating with Other Programs (Virtual Terminals)* shows how one program can load another program and how the programs can interact with each other.
- *Chapter 14. Designing and Coding Sensor I/O Programs* describes digital and analog input/output and shows how to read and write to sensor I/O devices.
- *Chapter 15. Designing and Coding Graphic Programs* shows how to code the instructions that produce graphic messages and draw curves on a display terminal.
- *Chapter 16. Controlling Spooling from a Program* describes how a program can control printed output.
- *Chapter 17. Creating, Storage and Retrieving Program Messages* shows how to save storage or coding time by creating messages than can be used by more than one program.

How This Book is Organized *(continued)*

- *Chapter 18. Queue Processing* shows how to create queues, store data in queues, and retrieve data from queues.
- *Appendix A. Tape Labels* shows the layout of tape labels.
- *Appendix B. Interrupt Processing* describes the interrupts that occur when a program interacts with a terminal.
- *Appendix C. Static Screens and Device Considerations* provides more details on reading and writing static screens to a terminal.

Aids in Using This Book

This book provides the following aids to assist you in using this book:

- A glossary which defines abbreviations and terms
- 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, Version 4.0 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 this book.

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

Contents

Contents

Contents

Contents

Figures

Chapter 1. Getting Started

This chapter is intended for people who have never coded an EDL (Event Driven Language) program. It describes the steps necessary to develop and run a simple program on the Series/1. Specifically, this chapter shows you how to design, code, enter, compile, link-edit, and execute an Event Driven Language (EDL) program.

We will show you all these steps using a simple example program. You may choose to actually enter and run this program on your Series/1 to gain hands-on experience.

Each of the major steps in the development and execution of an EDL program is covered in greater detail later in this book. The following chart describes these steps and shows you where in this book the material is covered.

Write the source program	Write a source program that does such things as read data, manipulate data, and write data *(Chapter 2).*
Enter the source program	Enter the source program by using the Session Manager to build a data set *(Chapter 3).*
Compile the source program	Compile your source program (produce object code from source code) *(Chapter 4).*
Link-edit the program	Produce an executable load module *(Chapter 5).*
Run the program	Cause your program to run or "execute" *(Chapter 6).*

Find and fix errors	Use the $DEBUG utility or a task error exit routine to help you locate and correct any problems in your program *(Chapter 7)*.

If you are familiar with EDL and the EDX operating system, skip this chapter and go to Chapter 2.

Designing a Program

The first step in the development of any program is the design of the program. You must be able to describe what you want the program to accomplish.

Typically, a program reads some data, processes the data, and writes the results. The sample program we have chosen does all of these. The program requests that an operator enter a number at the terminal. That number is added to a storage area ten times, and the results are displayed on the terminal screen.

Here are some questions you should ask when you plan a program. We have shown how we answered those questions in our sample program.

Questions	In our program
Where is the data coming from and what form will it take?	The data is a number that the operator enters at the terminal.
What do you want to do with the data and in what order do you want to process the data?	The number that is entered from the terminal will be added ten times to a storage area that you define.
Where do you print or record the results?	The results are displayed on the terminal screen.

In the next section, we will show you how to implement this design in an EDL program.

Coding the program

On the next few pages, we will show you how the design of this program was implemented. We will build the program step by step. We will not describe *every* possible operand of the instructions we use. (Operands for every EDL instruction are fully described in the *Language Reference*.)

EDL instructions and statements have the general format:

```
label          operation               operands
```

where these terms have the following meanings:

label The name you assign an instruction or statement. You can use this name in your program to refer to that specific instruction or statement. In most cases, the label is optional. Labels must begin in column 1; must begin with a letter or one of the special characters $, #, or @; and must be 1 to 8 characters long.

operation The name of the instruction or statement you are coding. The operation can begin in column 2 and cannot extend beyond column 71.

operands The data that is required to do an operation, or information on how the system is to perform the operation.

To continue a line of code on the next line, place any nonblank character in column 72 and continue the next line in column 16.

Starting the program

Any EDL program begins with the **PROGRAM** statement.

A **PROGRAM** statement defines the address or label of the first instruction to be executed. The PROGRAM statement also defines the name of the primary task of the program. (EDL programs may consist of multiple tasks. In our sample program, the primary task is the only task of the program.)

Our program statement looks like this:

```
ADD10     PROGRAM     STPGM
```

ADD10 is the *task name* of the primary (and only) task.

STPGM is the label of the first instruction to be executed.

Coding the program *(continued)*

Defining Your Data

The program needs two data areas: one to hold the input and one to hold the results of the process. Use the **DATA** statement to reserve storage for data.

```
ADD10     PROGRAM    STPGM
            .
            .
            .
COUNT     DATA       F'0'
SUM       DATA       F'0'
```

These DATA statements indicate that the reserved areas are type F (for fullword) and that the initial value of the areas is 0.

Since DATA statements do not cause any action to occur, place them either before the first instruction or after the last instruction.

Retrieving Data

The next step is to get input data into the program. In this program, we use a GETVALUE instruction to get the data.

```
ADD10     PROGRAM    STPGM
STPGM     GETVALUE   COUNT,'ENTER NUMBER: '

            .
            .
COUNT     DATA       F'0'
SUM       DATA       F'0'
```

When the GETVALUE instruction executes, the message "ENTER NUMBER: " appears on the terminal screen. When someone enters a number and presses the ENTER key, the system stores the number in the data area called **COUNT**.

Processing the Data

This program is going to add the number that is entered from the terminal to the contents of storage area SUM. You need an ADD instruction to perform the addition. The number is going to be added to COUNT ten times. So the ADD instruction is placed inside a DO loop, which consists of a DO instruction and an ENDDO instruction. The DO instruction indicates how many times the instructions (in this case, an ADD instruction) is to be executed.

```
ADD10       PROGRAM    STPGM
STPGM       GETVALUE   COUNT,'ENTER NUMBER: '
LOOP        DO         10,TIMES
              ADD      SUM,COUNT
            ENDDO
               .
               .
               .
COUNT       DATA        F'0'
SUM         DATA        F'0'
```

Obtaining the Results

At this point, the program includes instructions to read data and process the data. To print the results, you use two instructions: **PRINTEXT** and **PRINTNUM**.

```
ADD10       PROGRAM    STPGM
STPGM       GETVALUE   COUNT,'ENTER NUMBER: '
LOOP        DO         10,TIMES
              ADD      SUM,COUNT
            ENDDO
            PRINTEXT   '@RESULT='
            PRINTNUM   SUM
               .
               .
               .
COUNT       DATA       F'0'
SUM         DATA       F'0'
```

The **PRINTEXT** instruction will print "RESULT=" on the terminal screen. The "@" symbol will cause "RESULT=" to be printed on a new line on the terminal screen. The **PRINTNUM** instruction will print the results of the process, which is stored in the SUM data area.

Ending the Program

The program needs three more statements to be complete. The **PROGSTOP** statement stops the program. You code PROGSTOP after the last executable instruction in the program.

All EDL programs must end with the **ENDPROG** and **END** statements.

The completed program looks like this:

```
ADD10       PROGRAM    STPGM
STPGM       GETVALUE   COUNT,'ENTER NUMBER: '
LOOP        DO         10,TIMES
              ADD       SUM,COUNT
            ENDDO
            PRINTEXT   '@RESULT='
            PRINTNUM    SUM
            PROGSTOP
COUNT       DATA        F'0'
SUM         DATA        F'0'
            ENDPROG
            END
```

Coding the program *(continued)*

The next step is to enter your program into a data set. The instructions and statements that make up a program are called the *source program*. We will show you how to use the *session manager* to enter the source program. The session manager provides a series of menus to help you enter a source program.

Entering the Source Program into a Data Set

All the steps for entering the source program are listed below. If you want to actually enter the sample source program into a data set, follow the numbered steps.

To invoke the session manager on your terminal:

1. Press the attention key.

2. Type $L $SMMAIN.

3. Press the enter key.

When you press the enter key, the logon screen appears:

```
$SMMLOG: THIS TERMINAL IS LOGGED ON TO THE SESSION MANAGER-------------
                                                               09:55:31
ENTER 1-4 CHAR USER ID ==>                                     10/24/82
(ENTER LOGOFF TO EXIT)

ALTERNATE SESSION MENU ==>
(OPTIONAL)
```

To begin a session:

1. Type a unique user identification (called a *user ID*). The user id can be one to four characters long.

2. Press the enter key.

This chapter uses **ABCD** as the user ID.

The Primary Option Menu appears on the screen. To enter a source program into a data set, select option 1 (TEXT EDITING).

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ----------------------------
ENTER/SELECT PARAMETERS:                                  PRESS PF3 TO EXIT

                                                      10:00:00
      SELECT OPTION ==>  1                            10/24/82
                                                      ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

1. Type 1 on the SELECT OPTION line.

2. Press the enter key.

The $FSEDIT PRIMARY OPTION MENU appears on the screen. Use option 2 (EDIT) to create a new data set.

```
$FSEDIT PRIMARY OPTION MENU  --------------------------------STATUS = INIT
                                                          PRESS PF3 TO EXIT
OPTION ===>  2

DATASET NAME =========>          (CURRENTLY IN WORK DATASET)
VOLUME NAME ==========>

HOST DATASET ========>

ENTER A VOLUME NAME AND PRESS ENTER FOR A DIRECTORY LIST.

1 ---- BROWSE
2 ---- EDIT
3 ---- READ (HOST/NATIVE)
4 ---- WRITE (HOST/NATIVE)
5 ---- SUBMIT
6 ---- PRINT
7 ---- MERGE
8 ---- END
9 ---- HELP
```

Entering the Source Program into a Data Set *(continued)*

1. Type   on the OPTION line.

2. Press the enter key.

Your data set then appears. This is where you will type the source program.

```
EDIT --- $SMEABCD , EDX002      0( 1089)--------------------- COLUMNS 001 072
COMMAND INPUT ===>                                            SCROLL ==> HALF
***** ***** TOP OF DATA *****************************************************
.....
***** ***** BOTTOM OF DATA **************************************************
```

To enter the source program, do the following:

1. Type the first line of code.

2. Press the enter key to cause a blank entry line to appear.

3. Type the next line of code.

4. Press the enter key.

5. Repeat steps 3 and 4 until you have entered the entire source program.

6. When you finish entering the source program, move the cursor to the COMMAND INPUT line and type M (for "menu").

7. Press the enter key.

```
EDIT --- $SMEABCD , EDX002      0( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===> M                                          SCROLL ==> HALF
***** ***** TOP OF DATA *******************************************************
00010 ADD10        PROGRAM       STPGM
00020 STPGM        GETVALUE      COUNT,'ENTER NUMBER: '
00030 LOOP         DO            10,TIMES
00040                ADD         SUM,COUNT
00050              ENDDO
00060              PRINTEXT      '@RESULT='
00070              PRINTNUM      SUM
00080              PROGSTOP
00090 COUNT        DATA          F'0'
00100 SUM          DATA          F'0'
00110              ENDPROG
00120              END
***** ***** BOTTOM OF DATA ****************************************************
```

The $FSEDIT PRIMARY OPTION MENU appears again.

The next step is to write the data set to a volume. When you write the data set, you copy the data set from the temporary data set that $FSEDIT has been using. The data set name we have chosen is ADD10 and the volume name is EDX002. Select option 4 (WRITE) to write the data set to a volume.

```
$FSEDIT PRIMARY OPTION MENU  ----------------------------STATUS = MODIFIED
                                                         PRESS PF3 TO EXIT
OPTION ===> 4

DATASET NAME =========> ADD10      (CURRENTLY IN WORK DATASET)
VOLUME NAME ==========> EDX002

HOST DATASET ========>

ENTER A VOLUME NAME AND PRESS ENTER FOR A DIRECTORY LIST.

1 ---- BROWSE
2 ---- EDIT
3 ---- READ (HOST/NATIVE)
4 ---- WRITE (HOST/NATIVE)
5 ---- SUBMIT
6 ---- PRINT
7 ---- MERGE
8 ---- END
9 ---- HELP
```

1. Type 4 on the OPTION line.

2. Type ADD10 on the DATASET NAME line.

3. Type EDX002 on the VOLUME NAME line.

4. Press the enter key.

The prompt:

```
WRITE TO ADD10 ON EDX002 (Y/N)?
```

appears on the bottom of the screen. Type Y and press the enter key.

The message:

```
12 LINES WRITTEN TO ADD10    ,EDX002
```

appears on the bottom of the screen. This message means that your source program is 12 lines long and has been written to volume EDX002.

Entering the Source Program into a Data Set *(continued)*

Now that you have entered and written the source program to a data set, return to the Session Manager Primary Option Menu.

```
$FSEDIT PRIMARY OPTION MENU -----------------------------STATUS = SAVED
                                                         PRESS PF3 TO EXIT
OPTION ===>8

DATASET NAME =========>                (CURRENTLY IN WORK DATASET)
VOLUME NAME ==========>

HOST DATASET ========>

ENTER A VOLUME NAME AND PRESS ENTER FOR A DIRECTORY LIST.

1 ---- BROWSE
2 ---- EDIT
3 ---- READ (HOST/NATIVE)
4 ---- WRITE (HOST/NATIVE)
5 ---- SUBMIT
6 ---- PRINT
7 ---- MERGE
8 ---- END
9 ---- HELP
```

1. Type 8 on the OPTION line.

2. Press the enter key.

Compiling Your Source Program

To prepare a source program to run on the system, you must compile it into object code. To do this, use $EDXASM, the EDX compiler.

Before you actually begin to compile, you must allocate a data set to hold the output (the object code). Start by selecting option 3 (DATA MANAGEMENT).

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ------------------------
ENTER/SELECT PARAMETERS:                                 PRESS PF3 TO EXIT

                                                                10:42:07
       SELECT OPTION ==> 3                                      10/24/82
                                                                ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

1. Type **3** on the SELECT OPTION line.
2. Press the enter key.

Compiling Your Source Program *(continued)*

The Data Management Option Menu appears on the screen. To allocate your object code data set, you select option 1 ($DISKUT1).

```
$SMM03  SESSION MANAGER DATA MANAGEMENT OPTION MENU----------------------
ENTER/SELECT PARAMETERS:                                    PRESS PF3 TO RETURN

  SELECT OPTION ==> 1

   1 - $DISKUT1 (DISK(ETTE) ALLOCATE, LIST DIRECTORY)
   2 - $DISKUT2 (DISK(ETTE) DUMP/LIST DATASETS)
   3 - $COPYUT1 (DISK(ETTE) COPY DATASETS/VOLUMES)
   4 - $COMPRES (DISK(ETTE) COMPRESS A VOLUME)
   5 - $COPY    (DISK(ETTE) COPY DATASETS/VOLUMES)
   6 - $DASDI   (DISK(ETTE) SURFACE INITIALIZATION)
   7 - $INITDSK (DISK(ETTE) INITIALIZE/VERIFY)
   8 - $MOVEVOL (COPY DISK VOLUME TO MULTI-DISKETTES)
   9 - $IAMUT1  (INDEXED ACCESS METHOD UTILITY PROGRAM)
  10 - $TAPEUT1 (TAPE ALLOCATE, CHANGE, COPY)
  11 - $HXUT1   (H-EXCHANGE DATASET UTILITY)

   WHEN ENTERING THESE UTILITIES, THE USER IS EXPECTED
   TO ENTER A COMMAND. IF A QUESTION MARK (?) IS ENTERED
   INSTEAD OF A COMMAND, THE USER WILL BE PRESENTED WITH
   A LIST OF AVAILABLE COMMANDS.
```

1. Type 1 on the SELECT OPTION line.

2. Press the enter key.

The $DISKUT1 utility prompts you for the command and for information about the data set you want to create. Use the AL (allocate) command. Call the data set that will hold the object code ADDOBJ. Allocate a 25-record data set and use the default data type.

```
LOADING $DISKUT1      59P,11:00:00, LP=9200, PART= 1

$DISKUT1 - DATA SET MANAGEMENT UTILITY I

USING VOLUME EDX002

COMMAND (?): AL
MEMBER NAME: ADDOBJ
HOW MANY RECORDS? 25
DEFAULT TYPE = DATA - OK (Y/N)? Y
ADDOBJ CREATED

COMMAND  (?): EN
```

1. Type AL on the COMMAND (?) line.
2. Press the enter key.
3. Type ADDOBJ on the MEMBER NAME line.
4. Press the enter key.
5. Type 25 next to the HOW MANY RECORDS? prompt.
6. Press the enter key.
7. Type Y next to the DEFAULT TYPE = DATA - OK (Y/N)? prompt.
8. Press the enter key.

A message appears telling you that the ADDOBJ data set has been created. Enter the EN (end) command to return to the Data Management Option Menu screen.

1. Type EN next to the COMMAND (?) prompt.
2. Press the enter key.

The next step is to return to the Session Manager Primary Option Menu to begin the compile. To return to that menu, press the PF3 key.

Compiling Your Source Program *(continued)*

From the Session Manager Primary Option Menu, select option 2 (PROGRAM PREPARATION) to begin the compile step.

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ----------------------------
ENTER/SELECT PARAMETERS:                                      PRESS PF3 TO EXIT

                                                              11:12:07
       SELECT OPTION ==> 2                                    10/24/82
                                                              ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

1. Type 2 on the SELECT OPTION line.

2. Press the enter key.

The Program Preparation Option Menu appears on your screen. To compile the source program, select option 1 ($EDXASM COMPILER).

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU-------------------
ENTER/SELECT PARAMETERS:                                    PRESS PF3 TO RETURN

       SELECT OPTION ==> 1

            1 - $EDXASM COMPILER
            2 - $EDXASM/$EDXLINK
            3 - $S1ASM ASSEMBLER
            4 - $COBOL COMPILER
            5 - $FORT FORTRAN COMPILER
            6 - $PLI COMPILER/$EDXLINK
            7 - $EDXLINK LINKAGE EDITOR
            8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISORS
            9 - $UPDATE
           10 - $UPDATEH (HOST)
           11 - $PREFIND
           12 - $PASCAL COMPILER/$EDXLINK
           13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

1. Type 1 on the SELECT OPTION line.

2. Press the enter key.

The $EDXASM Parameter Input Menu appears on your screen. You must enter the name of your source program (data set ADD10 on volume EDX002) and your object output (data set ADDOBJ on volume EDX002).

```
$SMM0201: SESSION MANAGER $EDXASM PARAMETER INPUT MENU-----------------------
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN


  SOURCE INPUT   (NAME,VOLUME) ==> ADD10,EDX002

  OBJECT OUTPUT  (NAME,VOLUME) ==> ADDOBJ,EDX002

  OPTIONAL PARAMETERS ==>
  (SELECT FROM THE LIST BELOW)
--------------------------------------------------------
AVAILABLE PARAMETERS:     ABBREVIATION:          DESCRIPTION:
 NOLIST                    NO                     USED TO SUPPRESS LISTING
 LIST TERMINAL-NAME        LI TERMINAL-NAME       USE LIST * FOR THIS TERMINAL
 ERRORS TERMINAL-NAME      ER TERMINAL-NAME       USE ERRORS * FOR THIS TERMINAL
 CONTROL DATA SET,VOLUME   CO DATA SET,VOLUME     $EDXASM LANGUAGE CONTROL DATASET
 OVERLAY #                 OV #                   # IS NUMBER OF AREAS FROM 1 TO 6

DEFAULT PARAMETERS:
 LIST $SYSPRTR CONTROL $EDXL,ASMLIB OVERLAY 4
```

1. Type ADD10,EDX002 next to SOURCE INPUT (NAME,VOLUME).
2. Type ADDOBJ,EDX002 next to OBJECT OUTPUT (NAME,VOLUME).
3. Press the enter key.

$EDXASM then compiles the source program into object code and puts the object code into data set ADDOBJ. This data set is used as input to the next step, "Creating a Load Module."

The information listed under DEFAULT PARAMETERS means that the compiler will print a listing of the program on the system printer, $SYSPRTR.

Compiling Your Source Program *(continued)*

As the compilation runs, the following appears on your screen.

```
LOADING $JOBUTIL    4P,11:21:25, LP= 9400, PART= 1
REMARK
ASSEMBLE ADD10,EDX002 TO ADDOBJ,EDX002
*** JOB - $EDXASM - STARTED AT 11:21:56 00/00/00 ***

JOB      $EDXASM  ($SMP0201)  USERID=ABCD
LOADING $EDXASM      78P,11:22:28, LP= 9800, PART= 1

ASSEMBLY STARTED   1 OVERLAY  AREA  ACTIVE
COMPLETION CODE =          -1

$EDXASM  ENDED AT 11:22:55

$JOBUTIL ENDED AT 11:22:56

PRESS ENTER KEY TO RETURN
```

If the screen gets filled up before displaying PRESS ENTER KEY TO RETURN, press the enter key.

A completion code of -1 means that your compilation completed successfully. Any completion code other than -1 means the program did not compile successfully.

Checking Your Compiler Listing

The compiler prints a listing that consists of statistics, source code statements and object code, undefined or external symbols, and a completion code.

If you do not receive a completion code of -1, check your listing for errors, fix them in your source data set, and rerun the compilation. For information on fixing compiler errors, see "Checking Your Compiler Listing and Correcting Errors" on page PG-76.

If you receive a completion code of -1, do the following:

1. Press the enter key to return to the $EDXASM Parameter Input Menu.

2. Press the PF3 key to return to the Program Preparation Option Menu.

Creating a Load Module

The last step is creating a load module. The load module runs or "executes" on the system. In this example, we use the linkage editor, $EDXLINK, to create the load module. $EDXLINK LINKAGE EDITOR is option 7 on the Program Preparation Option Menu.

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU---------------------
ENTER/SELECT PARAMETERS:                                  PRESS PF3 TO RETURN

      SELECT OPTION ==> 7

            1 - $EDXASM COMPILER
            2 - $EDXASM/$EDXLINK
            3 - $S1ASM ASSEMBLER
            4 - $COBOL COMPILER
            5 - $FORT FORTRAN COMPILER
            6 - $PLI COMPILER/$EDXLINK
            7 - $EDXLINK LINKAGE EDITOR
            8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISORS
            9 - $UPDATE
           10 - $UPDATEH (HOST)
           11 - $PREFIND
           12 - $PASCAL COMPILER/$EDXLINK
           13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

1. Type 7 on the SELECT OPTION line.

2. Press the enter key.

The $EDXLINK Parameter Input Menu appears on your screen. Enter an asterisk (*) next to EXECUTION PARM to indicate that you want the system to prompt you for linkage editor statements.

```
$SMM0207: SESSION MANAGER $EDXLINK PARAMETER INPUT MENU---------------------
ENTER/SELECT PARAMETERS:                                  PRESS PF3 TO RETURN



  EXECUTION PARM ==> *

  ENTER A CONTROL DATA SET NAME,VOLUME OR
  AN ASTERISK (*) FOR INTERACTIVE MODE.

  OUTPUT DEVICE (DEFAULTS TO $SYSPRTR) ==>
```

1. Type * on the EXECUTION PARM line.

2. Press the enter key.

$EDXLINK displays the following screen:

```
LOADING $JOBUTIL    4P,11:27:06, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 11:27:16 11/13/82 ***

JOB      $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK      89P,11:27:18, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
  DEFAULT VOLUME = EDX002

STMT (?):
```

Next, enter an INCLUDE statement to indicate which object module to use. (Remember, the object module is ADDOBJ.) Then, enter a LINK statement to indicate the name of the output data set. When you enter the name of this data set (in this case, ADDPGM), the system allocates the data set.

1. Type INCLUDE ADDOBJ,EDX002 next to STMT (?).

2. Press the enter key.

3. Type LINK ADDPGM,EDX002 next to STMT (?).

4. Press the enter key.

After the system indicates that the link-edit is successful, return to the Primary Option Menu to execute your program. To return to the Primary Option Menu:

1. Type EN next to STMT (?).

2. Press the enter key.

3. Press the PF3 key to return to the Program Preparation Option Menu.

4. Press the PF3 key again.

Running Your Program

To run (or execute) your program, select option 6 (EXEC PROGRAM/UTILITY).

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ----------------------------
ENTER/SELECT PARAMETERS:                                   PRESS PF3 TO EXIT

                                                           11:42:07
      SELECT OPTION ==> 6                                  10/24/82
                                                           ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

1. Type 6 on the SELECT OPTION line.

2. Press the enter key.

The Execute Program/Utility menu appears. You must enter the program name (ADDPGM) and volume (EDX002). Then, type asterisks (*) next to the data sets not used.

```
$SMM06  SESSION MANAGER EXECUTE PROGRAM/UTILITY----------------------------
ENTER/SELECT PARAMETERS:                                   PRESS PF3 TO RETURN

 PROGRAM/UTILITY (NAME,VOLUME) ==> ADDPGM,EDX002

 PARAMETERS ===>

 DATA SET 1 (NAME,VOLUME / * = DS1 NOT USED) ===> *
 DATA SET 2 (NAME,VOLUME / * = DS2 NOT USED) ===> *
 DATA SET 3 (NAME,VOLUME / * = DS3 NOT USED) ===> *

 NOTE: IF A DATA SET (DS1, DS2 OR DS3) IS NOT USED,
       AN ASTERISK (*) MUST BE ENTERED IN THE DATA SET FIELD.
```

1. Type ADDPGM,EDX002 next to PROGRAM/UTILITY (NAME,VOLUME).

2. Type an asterisk in the DATA SET 1, DATA SET 2, and DATA SET 3 fields.

3. Press the enter key.

Running Your Program *(continued)*

The following text appears on the terminal:

```
LOADING $JOBUTIL    4P,11:48:21, LP= 9400, PART= 1
REMARK
EXECUTE  PROGRAM/UTILITY: ADDPGM
*** JOB - ADDPGM - STARTED AT 11:48:22 11/14/82 ***

JOB      ADDPGM ($SMP06) USERID=ABCD
LOADING ADDPGM        2P,11:48:23, LP= 9800, PART= 1
ENTER NUMBER:
```

The program displays ENTER NUMBER on the screen and waits for you to enter a number. (Remember that "ENTER NUMBER" was coded on the GETVALUE instruction.)

1. Type 5 next to ENTER NUMBER.

2. Press the enter key.

```
LOADING ADDPGM        2P,11:48:55, LP= 9800, PART= 1
ENTER NUMBER: 5
      RESULT=     50
ADDPGM   ENDED AT 11:48:57

$JOBUTIL ENDED AT 11:48:58

PRESS ENTER KEY TO RETURN
```

The program displays the results of the processing. The program:

1. Stored the number you entered (5) in an area called COUNT.

2. Added the value of COUNT to the value of SUM, which was initialized to 0.

3. Added the two values 10 times.

4. Displayed the result (RESULT=   50) on the terminal screen.

The PRINTEXT instruction displayed RESULT=. The PRINTNUM instruction displayed the value of SUM (50).

Chapter 2. Writing a Source Program

This chapter tells how to use the EDL instructions to handle the basic functions of the language: reading and writing data, data conversions, and data manipulation (such as moving, adding, and subtracting.)

This chapter discusses the following topics:

- Beginning the program
- Reserving storage
- Reading data into a data area
- Moving data
- Converting data
- Manipulating data
- Writing data from a data area
- Controlling program logic
- Ending the program

All the instructions are discussed in detail in the *Language Reference*. This chapter lists the instructions by function and discusses only a subset of them.

Writing a Source Program

Beginning the Program

The first statement in every EDL program must be a PROGRAM statement. The PROGRAM statement defines several things about the program to the Event Driven Executive, only two of which are discussed in this section.

Defining the Primary Task

Two important functions of the PROGRAM statement are to define the "primary task" and provide the label of the first "executable instruction."

The *primary task* is the first task the system starts when you invoke the program.

An *executable* instruction causes some action to take place. For example, instructions that read, write, move, or perform arithmetic operations are executable instructions.

The following example shows a program with task name TASK1. Its first executable instruction is at location START1.

```
TASK1          PROGRAM       START1
```

Identifying Data Sets to be Used in Your Program

Another important function of the PROGRAM statement is to identify the data sets that a program will use.

The DS= keyword operand of the PROGRAM statement allows you to idenfity up to nine data sets that the program can use. A *keyword operand* usually contains an equal (=) sign. The "keyword" to the left of the equal sign identifies what information you are supplying. The keyword operand must appear, of course, exactly as the system expects it. For example, if you code the DS= operand as SD=, the system would not recognize it. The advandage of keyword operands is that you can code them in any order.

When you specify data set names in the PROGRAM statement, the system opens the data sets when you load the program.

When the program executes, all data sets must already exist. One way to allocate data sets is with the $DISKUT1 utility.

If a program uses one data set *and* the data set resides on the IPL volume, the PROGRAM statement might look like this:

```
UPDATE   PROGRAM    START1,DS=TRANS
```

The program uses data set TRANS on the IPL volume.

If a program uses more than one data set and the data sets all reside on the IPL volume, the DS= operand would contain one set of parentheses as follows:

```
UPDATE   PROGRAM    START1,DS=(TRANS,MASTIN,MASTOUT)
```

The program uses data sets TRANS, MASTIN, and MASTOUT on the IPL volume.

If the data resides on a volume other than the IPL volume, two sets of parentheses are required. For example:

```
TASK1    PROGRAM    START1,DS=((DATA1,MYVOL),MASTER)
```

The program uses data set DATA1 on volume MYVOL and data set MASTER on the IPL volume.

Reserving Storage

This section shows how to reserve storage for arithmetic values or character strings.

EDL allows you to define arithmetic values in two ways: as "integer" data and as "floating-point" data. *Integer* data consists of positive and negative numbers with no decimal point. *Floating-point* data consists of positive and negative numbers that can have decimal points.

For example, you can define the number 7 as either a floating-point number or an integer. To define the number 7.5, however, you must define it as a floating-point number.

Reserving Storage for Integers

To reserve storage for an integer, you can use either the DATA or DC statement. The following DATA statement, for example, defines a storage area for a 2-byte signed integer.

```
NODOGS  DATA  F'0'
```

NODOGS is the name or label of the storage area. This type of storage area is often called a variable. The F defines a fullword (two bytes) and '0' assigns an initial value of zero to the area.

To set up more than one 1-word area in one statement, you can use the duplication factor. The statement:

```
FITABLE  DATA  15F'0'
```

reserves fifteen 1-word areas and assigns a zero to each.

You can use the areas called NODOGS and FITABLE in data manipulation instructions such as ADD and SUBTRACT.

Assigning an Initial Value

To assign an initial value, enclose the value in apostrophes as follows:

```
FIM      DATA  F'5280'
```

The storage area called FIM will contain the decimal value 5280 throughout the execution of your program, unless you change it.

You can also assign a hexadecimal value to a storage area. For example:

```
XFIM     DATA  X'14A0'
```

XFIM contains the hexadecimal value '14A0' (decimal 5280).

Defining a Halfword or Doubleword Data Area

You can also define a halfword (1-byte) or doubleword (4-byte) data area. The following statements reserve storage for halfword integers:

```
MSIX     DATA  H'-6'
SHVAR    DATA  H'0'
```

MSIX contains the value of minus 6.

To reserve four bytes of storage, define a data area as follows:

```
QTRMIL   DATA  D'250000'
LNGVAR   DATA  D'0'
```

QTRMIL occupies a doubleword (4 bytes) of storage and contains an initial value of 250,000 (decimal).

Defining Floating-Point Values

To define floating-point values, you can use either the DATA or DC statement. How large the number is determines how you define the storage. If the number falls between 10^{-76} and 10^{76} and contains less than seven significant digits, you can define a single-precision floating-point data area. Each single-precision floating-point number requires 4 bytes of storage.

The following DATA statement defines a storage area for a single-precision floating-point number.

```
NETPAY   DATA  E'000.00'
```

NETPAY is the name of the storage area. The E defines a floating-point data area and assigns it an initial value of zero.

To set up more than one floating-point data area, you can use the duplication factor. The statement

```
NPTAB    DATA  12E'000.00'
```

reserves storage for twelve 4-byte floating-point data areas and assigns an initial value of zero to each.

Assigning an Initial Value

To assign an intial value to a floating point data area, enclose the value in apostrophes as follows:

```
PI     DATA  E'3.14159'
```

PI contains the decimal value 3.14159. You can also express the exponent for a floating-point data area as in the following examples:

```
PIE    DATA  E'.314159E1'
PIE2   DATA  E'314.159E-2'
```

Defining an Extended-Precision Data Area

If a floating-point number requires more than 6 and fewer than 15 significant digits, you must use extended-precision floating point. Each extended-precision floating-point number requires 8 bytes of storage.

The following DATA statements define storage areas for extended-precision floating-point numbers:

```
MSMNT     DATA  L'0.000'
MYCELLS   DATA  L'15063842E12'
```

Defining Character Strings

To define character strings, you can use either the DATA or DC statement. The following DATA statement defines a storage area for a 6-byte character string:

```
NAME   DATA  C'TILTON'
```

NAME is the name or label of the storage area. The length of the storage area is the number of characters inside the apostrophes.

If you want an area of blanks, you can use the duplication factor:

```
BLNKS  DATA  10C' '
```

BLNKS is an area of 10 blanks.

To set up an area that contains a character string followed by blanks, define the storage area like this:

```
DOLCON  DATA  CL4'$$'
```

DOLCON contains two dollar signs ($$) followed by two blanks.

Reserving Storage *(continued)*

Assigning a Value to a Symbol

The EQU statement assigns a value to a symbol. You can use the symbol (the label on the EQU statement) as an operand in other instructions wherever symbols are allowed. If you use a label as an operand in an EQU statement, you must have defined it previously.

For example, you cannot code:

```
ABLE     EQU   BAKER
```

unless you have previously defined BAKER.

The following example assigns the word value X'0002' to A.

```
A        EQU    2
```

If you refer to the equated value with its label, the system assumes you are referring to a storage location. For example, if you use A in the following instruction:

```
         MOVE    B,A
```

the system moves the word at address 0002 to B.

If, however, you want to use the equated value as the *number* 2, you must precede the label with a plus sign (+) as follows:

```
        MOVE    B,+A
```

This instruction moves 2 to B.

The next example assigns the word value of A to B.

```
B        EQU    A
```

The following example shows how you can use the equated symbols in a program:

```
1           MOVE   C,A
2           MOVE   C,+A
3           MOVE   C,+B
4           MOVE   C,+A,(1,BYTE)
            .
            .
            .
5 A         EQU     2
6 B         EQU     A
    C         DATA    F
```

1 Move the contents of address 0002 to C.

2 Move X'0002' to C.

3 Move X'0002' to C.

4 Move the leftmost byte of the word value X'0002' (X'00') to C.

5 Define A with a word value of X'0002'.

6 Assign B the value of A (X'0002').

Defining an Input/Output Area

To define an area to read into or to write from, you must know where the data is coming from or where it is going.

If you are reading or writing data from tape, disk, or diskette, you can define an input/output area with a BUFFER statement, a DATA statement, or a DC statement.

If you are reading or writing data from a terminal, you can define an input/output area with a TEXT statement, a DATA statement, or a DC statement.

If you use either a DATA statement or a DC statement, however, you must precede the storage area with a word (2 bytes) containing the length and count. (Refer to the *Language Reference* for information on how the system constructs a storage area defined by a TEXT statement.)

Defining a BUFFER Statement

A BUFFER statement defines a data storage area. When you read or write records to disk, diskette, or tape, you can use the BUFFER statement to define the buffer. To define a 256-byte buffer, use the BUFFER statement as follows:

```
RDAREA   BUFFER   256,BYTES
```

RDAREA is the name of the buffer.

A buffer consists of an index, a length, and the data storage area. The index and the length each occupy one word (2 bytes). Therefore, a 256-byte buffer actually occupies 260 bytes of storage. For more information on the structure of a buffer, refer to the *Language Reference*.

Defining a TEXT Statement

Use the TEXT statement to define a message or storage area. Use the TEXT statement in conjunction with the PRINTEXT or READTEXT instructions. The PRINTEXT instruction prints the message or storage area on a terminal. The READTEXT instruction reads a character string from a terminal into the storage area defined by the TEXT statement.

When you code a TEXT statement, the system creates an area that consists of a 1-byte length, 1-byte count, and the message or storage area. Therefore, a 24-character message, for example, requires 26 bytes of storage. The maximum length of a TEXT statement is 254 bytes.

Reserving Storage *(continued)*

The following example creates the message ENTER YOUR NAME:

```
MSG1      TEXT      'ENTER YOUR NAME:'
```

To cause the message to appear on a terminal, code a PRINTEXT instruction that references MSG1, the name of the TEXT statement.

To define a storage area for data that you will read from a terminal, code the following:

```
ADDRESS   TEXT       LENGTH=30
```

A READTEXT instruction can read data from a terminal into the storage area by referencing ADDRESS, the name of the TEXT statement.

Reading Data into a Data Area

When you read data into a data area, the instruction you use depends on the kind of data and where it is coming from.

If the data resides on disk, diskette, or tape, use the READ instruction. If the data is coming from a terminal, use either the READTEXT or GETVALUE instruction. If the data is alphameric, use READTEXT. If the data consists of one floating-point number or one or more integers, use GETVALUE.

Reading Data from Disk or Diskette

You can read disk or diskette data sets either sequentially or directly. When you read, you always read a multiple of 256 bytes. In EDX, 256 bytes is called an "EDX record".

The READ instruction reads a record from one of the data sets you specify in the PROGRAM statement. The following READ instruction reads a record sequentially from the third data set defined on the PROGRAM statement.

```
          READ  DS3,DISKBUFF,1,0,ERROR=RDERROR,END=NOTFOUND
          .
          .
          .
DISKBUFF BUFFER  256,BYTES
```

The system reads one record (indicated by 1 in the third operand) sequentially (indicated by 0 in the fourth operand) into DISKBUFF. If no more records exist on the data set, the program branches to NOTFOUND. If an I/O error occurs, the program branches to RDERROR. Otherwise, the system places the data in the 256-byte buffer DISKBUFF.

To read a data set directly, code the fourth operand with an integer greater than zero as follows:

```
           READ   DS2,BUFR,1,52,ERROR=RDERR,END=ALLOVER
           .
           .
           .
BUFR       BUFFER  512,BYTES
```

The system reads the 52nd record (indicated by 52 in the fourth operand) into BUFR. If the data set does not contain 52 records, the program branches to ALLOVER. If an I/O error occurs, the program branches to RDERR. Otherwise, the system places one record (indicated by 1 in the third operand) into the 512-byte buffer BUFR.

Reading Data from Tape

You can read tape data sets sequentially only. A tape READ retrieves a record from 18 to 32,767 bytes long.

The following READ instruction reads a record from a tape.

```
           READ     DS1,BUFF,1,327,END=END1,ERROR=ERR,WAIT=YES
           .
           .
           .
BUFF       BUFFER   327,BYTES
```

The system reads one record (indicated by 1 in the third operand). The size of the record is 327 bytes (indicated by 327 in the fourth operand). If no more records exists on the data set, control transfers to END1. If an error occurs, control transfers to ERR. The system waits for the operation to complete before continuing (WAIT=YES). The buffer BUFF is 327 bytes long.

The following READ instruction reads 2 records into buffer BUFF2.

```
           READ      DS1,BUFF2,2,327,END=END1,ERROR=ERR,WAIT=YES
           .
           .
BUFF2      BUFFER    654,BYTES
```

The system reads two records (indicated by 2 in the third operand). The size of each record is 327 bytes (indicated by 327 in the fourth operand). If no more records exists on the data set, control transfers to END1. If an error occurs, control transfers to ERR. The system waits for the operation to complete before continuing (WAIT=YES). The buffer BUFF2 is 654 bytes long.

Reading from a Terminal

To read data that an operator enters on a terminal, you can use either the READTEXT or GETVALUE instruction. The READTEXT instruction allows you to read alphameric data (alphabetic characters, numbers, and special characters). With the GETVALUE instruction, you can read numbers (both integer and floating-point) only.

Reading Data into a Data Area *(continued)*

Reading Alphameric Data

To read an alphameric data item into a storage area, use the READTEXT instruction as follows:

```
          READTEXT COUNTY,'ENTER YOUR COUNTY: ',SKIP=1,MODE=LINE
          .
          .
  COUNTY  TEXT     LENGTH=20
```

The instruction displays the prompt **ENTER YOUR COUNTY**: and the system waits for a response. When the operator enters a name and presses the enter key, the system stores the text string in an area called COUNTY.

The operand SKIP=1 causes the system to skip one line before displaying the prompt. The operand MODE=LINE allows blanks in the response.

Unless you know how the system constructs a storage area defined by a TEXT statement, you should read into an area defined by a TEXT statement.

For more information on reading alphameric data from terminals, see Chapter 8, "Reading and Writing Data from Screens" on page PG-115.

Reading Numeric Data

The GETVALUE instruction allows you to read either a single floating-point value or more than one integer from a terminal. The following instruction reads a floating-point number:

```
       GETVALUE  BASAL,'ENTER YOUR BASE SALARY: ',             C
             TYPE=F,FORMAT=(6,2,F)
       .
       .
 BASAL DATA  E'0.00'
```

The instruction prompts the operator, waits for a response, reads the response, and stores the number in BASAL. You must have defined BASAL as a floating-point variable. The operand TYPE=F means that the number will be a single-precision floating-point number.

The operand FORMAT=(6,2,F) says that the number will occupy six positions on the screen (including the decimal point), that the number will contain two digits to the right of the decimal point, and that the number will be an "F-type" number such as 325.78.

To read more than one integer, code a third operand on the instruction as follows:

```
          GETVALUE HEIGHTS,'ENTER FIVE HEIGHTS (IN INCHES): ',5
```

The instruction assumes that you have defined HEIGHTS as follows:

```
  HEIGHTS  DATA  5F'0'
```

Moving Data

You can move data from one place in storage to another with the MOVE instruction. Unless you specify otherwise, the system moves one word (two bytes). For example, the instruction

```
         MOVE   OLDDATA,NEWDATA
         .
         .
OLDDATA  DATA   F'0'
NEWDATA  DATA   F'0'
```

moves the word at NEWDATA to OLDDATA. Note that whatever OLDDATA contained before the instruction was executed has been overlaid by the data in NEWDATA.

To move more than one word, you must code a third operand. For example, the following instruction moves 12 words from NEWNAME to OLDNAME:

```
         MOVE   OLDNAME,NEWNAME,12
         .
         .
OLDNAME  DATA   F'0'
NEWNAME  DATA   F'0'
```

To move bytes, code the third operand like this:

```
         MOVE   OLDADDR,NEWADDR,(15,BYTE)
         .
         .
OLDADDR  TEXT   LENGTH=15
NEWADDR  TEXT   LENGTH=15
```

This instruction moves the 15 bytes at NEWADDR to OLDADDR.

To move doublewords, code the third operand as follows:

```
         MOVE   OLDDESC,NEWDESC,(10,DWORD)
         .
         .
OLDDESC  DATA   10D'0'
NEWDESC  DATA   10D'0'
```

This instruction moves the 10 doublewords at NEWDESC to OLDDESC.

To move floating-point value, you must specify FLOAT (for single-precision) or DFLOAT (for extended-precision).

```
         MOVE   TEMPS,MSMNTS,(4,FLOAT)
         .
         .
TEMPS    DATA   4E'0.0'
MSMNTS   DATA   4E'0.0'
```

This instruction moves the four single-precision floating-point values at MSMNTS to TEMPS.

EDL allows you to do two types of conversion: from binary to an EBCDIC character string and from an EBCDIC character string to binary. The CONVTB instruction converts from binary to an EBCDIC character string, while the CONVTD instruction converts from an EBCDIC character string to binary.

Converting to an EBCDIC Character String

If a number has been stored as a binary number, you must convert it to an EBCDIC character string if, for example, you want to display the number with the PRINTEXT instruction.

A binary number is any variable you have defined as single-precision integer, double-precision integer, single-precision floating point, extended-precision floating point, or hexadecimal.

You must convert any of the following data items before you can display them:

```
NODOGS   DATA  F'0'
POPKANS  DATA  D'0'
PI       DATA  E'0.0'
FINMEAS  DATA  L'0.0'
XTRAS    DATA  X'0'
```

The following example converts a single-precision integer to an EBCDIC character string.

```
        CONVTB  DOGS,NODOGS,PREC=S,FORMAT=(5,0,I)
        .
        .
DOGS    TEXT    LENGTH=5
NODOGS  DATA    F'0'
```

The instruction converts the single-precision integer (indicated by PREC=S) in NODOGS and puts the result in DOGS. The FORMAT operand says that you want the converted output to be 5 digits long, contain 0 digits to the right of the decimal point, and be an integer (I).

To convert a double-precision integer, code the CONVTB instruction as follows:

```
        CONVTB  POP,POPKANS,PREC=D,FORMAT=(8,0,I)
        .
        .
POP     TEXT    LENGTH=8
POPKANS DATA    D'0'
```

The instruction converts the double-precision integer (indicated by PREC=D) in POPKANS and puts the result of the conversion in POP. The FORMAT operand says that you want the converted output to be 8 digits long, contain 0 digits to the right of the decimal point, and be an integer (I).

The following instruction converts a single-precision floating-point variable:

```
        CONVTB  PIOP,PI,PREC=F,FORMAT=(15,4,F)
        .
        .
PIOP    TEXT    LENGTH=16
PI      DATA    E'0.0000'
```

The instruction converts the single-precision floating-point variable (indicated by PREC=F) in PI and puts the result of the conversion in PIOP. The FORMAT operand says that you want the converted output to be 15 digits long, contain 4 digits to the right of the decimal point, and be a floating-point numeric (F).

To convert an extended-precision floating-point variable:

```
         CONVTB   FLOP,OP,PREC=L,FORMAT=(17,3,E)
         .
         .
FLOP     TEXT     LENGTH=24
OP       DATA     L
```

The instruction converts the extended-precision floating-point variable (indicated by PREC=L) in OP and puts the result of the conversion in FLOP. The FORMAT operand says that you want the converted output to be 17 digits long, contain 3 digits to the right of the decimal point, and be expressed in exponent notation (E).

Converting to Binary

If you read a number with the READTEXT instruction, you must convert it to binary before you can add, subtract, multiply, or divide.

The CONVTD instruction converts a character string to a binary number. You can convert a character string that contains a number to a single-precision integer, a double-precision integer, single-precision floating point, or extended-precision floating point.

The following CONVTD instruction converts a single-precision integer to binary:

```
         CONVTD   GNUS,NOGNUS,PREC=S,FORMAT=(5,0,I)
         .
         .
GNUS     DATA     F'0'
NOGNUS   TEXT     LENGTH=5
```

The instruction converts the EBCDIC character string in NOGNUS and puts the result in GNUS, a single-precision integer variable (indicated by PREC=S).

The FORMAT operand says that the data to be converted is 5 digits long, contains 0 digits to the right of the decimal point, and is an integer(I).

To convert a number that is greater than 32,767, you must convert it to a double-precision integer as follows:

```
         CONVTD   FLEAS,NOFLEAS,PREC=D,FORMAT=(9,0,I)
         .
         .
FLEAS    DATA     D'0'
NOFLEAS  TEXT     LENGTH=9
```

Converting Data *(continued)*

The instruction converts the EBCDIC character string in NOFLEAS and puts the result in FLEAS, a double-precision integer variable (indicated by PREC=D).

The FORMAT operand says that the data to be converted is 9 digits long, contains 0 digits to the right of the decimal point, and is an integer(I).

To convert to single-precision floating point, code the instruction as follows:

```
          CONVTD  AVTEMP,TEMP,PREC=F,FORMAT=(8,2,F)
          .
          .
AVTEMP    DATA    E'0.0'
TEMP      TEXT    LENGTH=9
```

The instruction converts the EBCDIC character string in TEMP and puts the result in AVTEMP, a single-precision floating-point variable (indicated by PREC=F).

The FORMAT operand says that the data to be converted is 8 digits long, contains 2 digits to the right of the decimal point, and is a floating-point number (F).

To convert to extended-precision floating point, code the instruction as follows:

```
          CONVTD  AVCOST,COST,PREC=L,FORMAT=(15,3,E)
          .
          .
AVCOST    DATA    L'0.00'
COST      TEXT    LENGTH=20
```

The instruction converts the EBCDIC character string in COST and puts the result in AVCOST, an extended-precision floating-point variable (indicated by PREC=L).

The FORMAT operand says that the data to be converted is 15 digits long, contains 3 digits to the right of the decimal point, and is expressed in exponent notation (E).

Converting from Floating Point to Integer

If you want to manipulate data, both operands in the operation must be either floating point or integer.

To convert a single-precision floating-point number to integer, code the FPCONV instruction as follows:

```
          FPCONV  INTNUM,FPNUM,PREC=SF
          .
          .
INTNUM    DATA    F'0'
FPNUM     DATA    E'0.0'
```

The instruction converts the single-precision floating-point number in FPNUM and puts the result in INTNUM, a single-precision integer variable. The PREC operand indicates that

INTNUM is a single-precision integer (S) and that FPNUM is a single-precision floating-point number (F).

To convert an extended-precision floating-point number to double-precision integer, code the FPCONV instruction as follows:

```
         FPCONV   INTDBL,FPEXT,PREC=DL
         .
         .
INTDBL   DATA     D'0'
FPEXT    DATA     L'0.0'
```

The instruction converts the extended-precision floating-point number in FPEXT and puts the result in INTDBL, a double-precision integer variable. The PREC operand indicates that INTDBL is a double-precision integer (D) and that FPEXT is an extended-precision floating-point number (L).

Note: When you convert from floating point to integer, remember that the system truncates all data to the right of the decimal point.

Converting from Integer to Floating Point

To convert a single-precision integer to floating-point, code the FPCONV instruction as follows:

```
         FPCONV   FPNUM,INTNUM,PREC=FS
         .
         .
INTNUM   DATA     F'0'
FPNUM    DATA     E'0.0'
```

The instruction converts the single-precision integer INTNUM and puts the result in FPNUM, a single-precision floating-point variable. The first letter in the PREC operand (F) indicates that FPNUM is a single-precision floating-point variable. The second letter (S) indicates that INTNUM is a single-precision integer.

To convert a double-precision integer to floating-point:

```
         FPCONV   FPEXT,INTDBL,PREC=LD
         .
         .
INTDBL   DATA     D'0'
FPEXT    DATA     L'0.0'
```

The instruction converts the double-precision integer INTDBL and puts the result in FPEXT, an extended-precision floating-point variable. The first letter in the PREC operand (L) indicates that FPEXT is an extended-precision floating-point variable. The second letter (D) indicates that INTDBL is a double-precision integer.

Converting Data *(continued)*

Checking for Conversion Errors

Each time you execute an instruction that converts data, the system expects the data to be numeric. If you try to convert a character other than a number, a conversion error occurs.

If, for example, a program prompts an operator for a number and he or she enters a letter, the system places a return code in the task code word. You can check for a conversion error as follows:

```
BEGIN    PROGRAM  START
         .
         .
         .
         CONVTD    GNUS,NOGNUS,PREC=S,FORMAT=(5,0,I)
ERRTEST  MOVE      TASKRC,BEGIN
         IF        (TASKRC,NE,-1),GOTO,CHECK
         ENDIF
         .
         .
         .
CHECK    PRINTEXT  'CONVERSION ERROR',SKIP=1
         PRINTNUM  TASKRC
         GOTO      END
         .
         .
         .
END      PROGSTOP
TASKRC   DATA      F'0'
GNUS     DATA      F'0'
NOGNUS   TEXT      LENGTH=5
         ENDPROG
         END
```

The instructions at label ERRTEST compare the return code of the CONVTD instruction with the successful return code (-1). IF NOGNUS contains a nonnumeric character, the system branches to CHECK.

You must test the return code before executing any other instruction because the system may overlay the task code word with the return code of the next instruction.

Manipulating Data

The data manipulation instructions perform arithmetic operations on single- or double-precision integers and single- or extended-precision floating-point numbers. You can also manipulate two bit-strings with logical instructions such as inclusive-OR and exclusive-OR.

Manipulating Integer Data

The instructions that manipulate integers add, subtract, multiply, or divide two integers. If two numbers are floating-point numbers, you must use floating-point instructions.

If one number is a floating-point number and the other is an integer, use the FPCONV instruction to convert one of the numbers to match the form of the other.

The instructions have the following general form:

```
operation    operand1,operand2
```

The flow of data is from *operand2* to *operand1*.

The ADD instruction adds the data in *operand2* to the data in *operand1* and places the results in *operand1*.

The SUBTRACT instruction subtracts the data in *operand2* from the data in *operand1* and places the results in *operand1*.

The DIVIDE and MULTIPLY instructions multiply or divide the data in *operand1* by the data in *operand2* and store the results in *operand1*.

Adding Integers

The ADD instruction adds two integers. If A and B are integers, you can add A to B with the following instruction:

```
ADD  B,A
```

The result of the addition replaces B. The value in A remains unchanged.

To add two integers without altering the first operand, use the RESULT operand as follows:

```
ADD  CAT,DOG,RESULT=GIRAFFE
```

The instruction adds DOG to CAT and places the result in GIRAFFE. The values in DOG and CAT remain unchanged.

Adding Double-Precision Integers

Unless you specify otherwise, EDL assumes that the integers are single-precision (1-word) integers. To add two double-precision (2-word) integers, specify the PREC operand as follows:

```
ADD  TOTVEG,BEETS,PREC=DD
```

The operand PREC=DD says that both TOTVEG and BEETS are double-precision integers.

If only one of the operands is a double-precision integer, it must be the first operand. In addition, if you specify the RESULT operand, it must be a double-precision variable. For example:

```
ADD  GHANA,CHAD,RESULT=TOTPOP,PREC=D
```

The operand PREC=D says that GHANA and TOTPOP are double-precision integers. The absence of the second letter (D or S) on the PREC operand means that CHAD is a single-precision integer.

Adding Consecutive Integers

To add more than one set of integers, you can specify the number of integers you want to add. For example:

```
ADD   NEWTOTS,OLDTOTS,10
```

The instruction adds the 1-word integer at OLDTOTS to NEWTOTS. Then the instruction adds the word in OLDTOTS+2 to the word at NEWTOTS+2. The instruction continues to add until it adds the word at OLDTOTS+18 to the word at NEWTOTS+18. This instruction, then, adds the 10 consecutive words at OLDTOTS to the 10 consecutive words at NEWTOTS. You can specify up to 32,767 consecutive additions.

Subtracting Integers

The SUBTRACT instruction subtracts one integer from another. If QUERY and ANSWER are integers, you can subtract ANSWER from QUERY with the following instruction:

```
SUBTRACT    QUERY,ANSWER
```

The result of the subtraction replaces QUERY. The value in ANSWER remains unchanged.

To subtract two integers without altering the first operand, use the RESULT operand as follows:

```
SUBTRACT    POOLS,STREAMS,RESULT=LAKES
```

The instruction subtracts STREAMS from POOLS and places the result in LAKES. The values in POOLS and STREAMS remain unchanged.

Subtracting Double-Precision Integers

Unless you specify otherwise, EDL assumes that the integers are single-precision (1-word) integers. To subtract two double-precision (2-word) integers, specify the PREC operand as follows:

```
SUBTRACT     TOTFRUT,PRUNES,RESULT=REST,PREC=DD
```

The instruction subtracts PRUNES from TOTFRUT and places the result in REST. The operand PREC=DD says that TOTFRUT, PRUNES, and REST are all double-precision integers.

If only one of the operands is a double-precision integer, it must be the first operand. In addition, if you specify the RESULT operand, it must be a double-precision variable. For example:

```
SUBTRACT     ATTEND,MALES,RESULT=FEMALES,PREC=D
```

The instruction subtracts MALES from ATTEND and places the result in FEMALES. The operand PREC=D says that ATTEND and FEMALES are double-precision integers. The absence of the second letter (D or S) on the PREC operand means that MALES is a single-precision integer.

Subtracting Consecutive Integers

To subtract more than one set of integers, you can specify the number of integers you want to subtract. For example:

```
SUBTRACT NEWTOTS,OLDTOTS,6
```

The instruction subtracts the 1-word integer at OLDTOTS from NEWTOTS. Then the instruction subtracts the word in OLDTOTS+2 from the word at NEWTOTS+2. The instruction continues to subtract until it subtracts the word at OLDTOTS+10 from the word at NEWTOTS+10. This instruction, then, subtracts the 6 consecutive words at OLDTOTS from the 6 consecutive words at NEWTOTS. You can specify up to 32,767 consecutive subtractions.

Multiplying Integers

The MULTIPLY instruction multiplies one integer by another.

If M and N are single-precision integers, you can multiply M by N as follows:

```
MULTIPLY   M,N
```

The result of the multiplication replaces M.

You can also multiply an integer by a constant. The following instruction multiplies FEET by the constant 12:

```
MULTIPLY   FEET,12
```

The result of the multiplication replaces FEET.

To multiply two integers without altering the first operand, use the RESULT operand as follows:

```
MULTIPLY  BOXES,WEIGHT,RESULT=TOTWGT
```

The instruction multiplies BOXES by WEIGHT and places the result in TOTWGT. The values in BOXES and WEIGHT do not change.

Multiplying Double-Precision Integers

Unless you specify otherwise, EDL assumes that integers are single-precision (1-word) integers. To multiply two double-precision (2-word) integers, specify the PREC operand as follows:

```
MULTIPLY GRAPES,PITS,RESULT=TOTPITS,PREC=DD
```

Manipulating Data *(continued)*

The instruction multiplies GRAPES by PITS and places the result in TOTPITS. The operand PREC=DD says that GRAPES, PITS, and TOTPITS are all double-precision integers.

If only one of the operands is a double-precision integer, it must be the first operand. In addition, if you specify the RESULT operand, it must be a double-precision variable. For example:

```
MULTIPLY ATTEND,GAMES,RESULT=TOTATT,PREC=D
```

The instruction multiplies ATTEND by GAMES and places the result in TOTATT. The operand PREC=D says that ATTEND and FEMALES are double-precision integers. The absence of the second letter (D or S) on the PREC operand means that GAMES is a single-precision integer.

Multiplying Consecutive Integers

To multiply more than one set of integers, you can specify the number of integers you want to multiply. For example:

```
MULTIPLY SALRIES,RATES,400
```

The instruction multiplies the 1-word integer at RATES by SALRIES and stores the result in SALRIES. Then the instruction multiplies the word in RATES+2 by the word at SALRIES+2. The instruction continues to multiply until it multiplies the word at RATES+798 by the word at SALRIES+798. This instruction, then, multiplies the 400 consecutive words at RATES by the 400 consecutive words at SALRIES. You can specify up to 32,767 consecutive multiplications.

Dividing Integers

The DIVIDE instruction divides one integer by another. The system places the remainder in the first word of the task control block (TCB).

If P and Q are single-precision integers, you can divide P by Q as follows:

```
DIVIDE   P,Q
```

The result of the division replaces P.

You can also divide an integer by a constant. The following instruction divides FEET by the constant 3:

```
DIVIDE   FEET,3
```

The result of the division replaces FEET.

To divide two integers without altering the first operand, use the RESULT operand as follows:

```
DIVIDE   TOTWGT,BOXES,RESULT=BOXWGT
```

The instruction divides TOTWGT by BOXES and places the result in BOXWGT. The values in TOTWGT and BOXES do not change.

Dividing Double-Precision Integers

Unless you specify otherwise, EDL assumes that integers are single-precision (1-word) integers. To divide double-precision (2-word) integers, specify the PREC operand as follows:

```
DIVIDE   TOTSAL,NOEMPS,RESULT=AVESAL,PREC=DD
```

The instruction divides TOTSAL by NOEMPS and places the result in AVESAL. The operand PREC=DD says that TOTSAL, NOEMPS, and AVESAL are all double-precision integers.

If only one of the operands is a double-precision integer, it must be the first operand. In addition, if you specify the RESULT operand, it must be a double-precision variable. For example:

```
DIVIDE   TOTATT,GAMES,RESULT=AVEATT,PREC=D
```

The instruction divides TOTATT by GAMES and places the result in AVEATT. The operand PREC=D says that TOTATT and AVEATT are double-precision integers. The absence of the second letter (D or S) on the PREC operand means that GAMES is a single-precision integer.

Dividing Consecutive Integers

To divide more than one set of integers, you can specify the number of integers you want to divide. For example:

```
DIVIDE   SALRIES,RATES,100
```

The instruction divides the 1-word integer at RATES by SALRIES. Then the instruction divides the word in RATES+2 by the word at SALRIES+2. The instruction continues to divide until it divides the word at RATES+198 by the word at SALRIES+198. This instruction, then, divides the 100 consecutive words at RATES by the 100 consecutive words at SALRIES. You can specify up to 32,767 consecutive divisions.

Accessing the Remainder

One way to access the remainder is to use the TCBGET instruction as in the following example:

```
        DIVIDE  SALRIES,RATES
        TCBGET  REMAIN,$TCBCO
        .
        .
REMAIN  DATA   F'0'
```

The instruction puts the first word of the task control block into REMAIN.

Manipulating Data *(continued)*

Manipulating Floating-Point Data

EDL allows you to add, subtract, multiply, and divide floating-point numbers. Floating-point numbers are positive and negative numbers that can have decimal points.

To use floating-point instructions, you must:

- Have the hardware floating-point feature installed on your system.
- Include floating-point support in the supervisor when it is generated.
- Specify FLOAT=YES on both the PROGRAM and TASK statements whenever you use floating-point instructions in any task within a program.
- Define the variables you are manipulating as floating-point variables.

Adding Floating-Point Data

The FADD instruction adds two floating-point numbers. If A and B are floating-point numbers, you can add A to B with the following instruction:

```
FADD    B,A
```

The result of the addition replaces B. The value in A remains unchanged.

To add two floating-point numbers without altering the first operand, use the RESULT operand as follows:

```
FADD    MYSAL,YOURSAL,RESULT=OURSALS
```

The instruction adds MYSAL to YOURSAL and places the result in OURSALS. The values in MYSAL and YOURSAL remain unchanged.

Adding Extended-Precision Floating-Point Numbers

Unless you specify otherwise, EDL assumes that the floating-point numbers are single-precision (2-word) floating-point numbers. To add two extended-precision (4-word) floating-point numbers, specify the PREC operand as follows:

```
FADD    TOTSAL,PRESAL,PREC=LL
```

The operand PREC=LL says that both TOTSAL and PRESAL are extended-precision floating-point numbers.

If only one of the operands is an extended-precision floating-point number, the PREC operand must reflect the precision. In the following example:

```
FADD    MSMNT1,MSMNT2,RESULT=MSMTS,PREC=LFL
```

The operand PREC=LFL says that MSMNT1 and MSMTS are extended-precision floating-point numbers and MSMNT2 is a single-precision floating-point number.

Subtracting Floating-Point Numbers

The FSUB instruction subtracts one floating-point number from another. If OCTEMP and NOVTEMP are floating-point numbers, you can subtract NOVTEMP from OCTEMP with the following instruction:

```
FSUB     OCTEMP,NOVTEMP
```

The result of the subtraction replaces OCTEMP. The value in NOVTEMP remains unchanged.

To subtract two floating-point numbers without altering the first operand, use the RESULT operand as follows:

```
FSUB     SAL,DEDUCS,RESULT=NET
```

The instruction subtracts DEDUCS from SAL and places the result in NET. The values in SAL and DEDUCS remain unchanged.

Subtracting Extended-Precision Floating-Point Numbers

Unless you specify otherwise, EDL assumes that the floating-point numbers are single-precision (2-word) floating-point numbers. To subtract two extended-precision (4-word) floating-point numbers, specify the PREC operand as follows:

```
FSUB    TOTSAL,TOTDUCS,RESULT=TOTNP,PREC=LLL
```

The instruction subtracts TOTDUCS from TOTSAL and places the result in TOTNP. The operand PREC=LLL says that TOTSAL, TOTDUCS, and TOTNP are all extended-precision floating-point numbers.

If only one of the operands is a extended-precision floating-point number, the PREC operand should reflect the precision. In the following example:

```
FSUB    SMALL,LARGE,RESULT=MINUS,PREC=FLF
```

The instruction subtracts LARGE from SMALL and places the result in MINUS. The operand PREC=FLF says that SMALL and MINUS are single-precision and that LARGE is an extended-precision floating-point number.

Multiplying Floating-Point Numbers

The FMULT instruction multiplies one floating-point number by another.

If M and N are single-precision floating-point numbers, you can multiply M by N as follows:

```
FMULT    M,N
```

The result of the multiplication replaces M.

You can also multiply a floating-point number by an integer constant. The following instruction multiplies FEET by the integer constant 12:

```
FMULT    FEET,12
```

The result of the multiplication replaces FEET.

To multiply two floating-point numbers without altering the first operand, use the RESULT operand as follows:

```
FMULT   LENGTH,WIDTH,RESULT=AREA
```

The instruction multiplies LENGTH by WIDTH and places the result in AREA. The values in LENGTH and WIDTH do not change.

Multiplying Extended-Precision Floating-Point Numbers

Unless you specify otherwise, EDL assumes that floating-point numbers are single-precision (2-word) floating-point numbers. To multiply two extended-precision (4-word) floating-point numbers, specify the PREC operand as follows:

```
FMULT   PI,DIAM,RESULT=CIRCUM,PREC=LLL
```

The instruction multiplies PI by DIAM and places the result in CIRCUM. The operand PREC=LLL says that PI, DIAM, and CIRCUM are all extended-precision floating-point numbers.

If only one of the operands is a double-precision floating-point number, the PREC operand must reflect the precision. The following example:

```
FMULT   BASEAREA,HEIGHT,RESULT=VOLUME,PREC=LFL
```

multiplies BASEAREA by HEIGHT and places the result in VOLUME. The operand PREC=LFL says that BASEAREA and VOLUME are extended-precision floating-point numbers and that HEIGHT is a single-precision floating-point number.

Dividing Floating-Point Numbers

The FDIVD instruction divides one floating-point number by another. The system places the remainder in the first word of the task control block (TCB).

If P and Q are single-precision floating-point numbers, you can divide P by Q as follows:

```
FDIVD    P,Q
```

The result of the division replaces P.

You can also divide a floating-point number by a constant. The following instruction divides FEET by the integer constant 3:

```
FDIVD   FEET,3
```

The result of the division replaces FEET.

To divide two floating-point numbers without altering the first operand, use the RESULT operand as follows:

```
FDIVD   TOTWGT,BOXES,RESULT=BOXWGT
```

The instruction divides TOTWGT by BOXES and places the result in BOXWGT. The values in TOTWGT and BOXES do not change.

Dividing Extended-Precision Floating-Point Numbers

Unless you specify otherwise, EDL assumes that floating-point numbers are single-precision (2-word) floating-point numbers. To divide two extended-precision (4-word) floating-point numbers, specify the PREC operand as follows:

```
FDIVD   CUBICFT,BASEAREA,RESULT=HEIGHT,PREC=LLL
```

The instruction divides CUBICFT by BASEAREA and places the result in HEIGHT. The operand PREC=LLL says that CUBICFT, BASEAREA, and HEIGHT are all extended-precision floating-point numbers.

If only one of the operands is an extended-precision floating-point number, the PREC operand must reflect the precision. The following example:

```
FDIVD   TOTSAL,NOEMPS,RESULT=AVESAL,PREC=LFL
```

divides TOTSAL by NOEMPS and places the result in AVESAL. The operand PREC=LFL says that TOTSAL and AVESAL are extended-precision floating-point numbers and that NOEMPS is a single-precision floating-point number.

Manipulating Logical Data

The instructions that manipulate logical data make a bit-by-bit comparison of two bit strings. The result of the comparison depends on the instruction.

The Exclusive-OR Instruction

The exclusive-OR instruction (EOR) compares two bit strings and produces a third bit string, called the resulting field.

The instruction compares the two bit strings one bit at a time. If the bits are the same, the instruction sets a bit in the resulting field to 0. If the bits are not the same, the instructions sets a bit in the resulting field to 1.

If the bit strings are identical, the resulting field contains all 0's. If one or more bits differ, the resulting field contains a mixture of 0's and 1's.

The following example compares PHI to CHI and places the result in PHI.

```
     EOR    PHI,CHI
```

The following table shows PHI and CHI before and after the instruction executes.

Data Item	Hex	Binary
PHI (before)	049C	0000 0100 1001 1100
CHI	56AB	0101 0110 1010 1011
PHI (after)	5237	0101 0010 0011 0111

To compare a variable to a constant, code *operand2* as follows:

```
     EOR    MU,X'5280'
```

The following table shows MU before and after the instruction executes.

Data Item	Hex	Binary
MU (before)	F0F0	1111 0000 1111 0000
constant	5280	0101 0010 1000 0000
MU (after)	A270	1010 0010 0111 0000

To compare two bit strings without altering the first operand, use the RESULT operand as follows:

```
     EOR    SIGMA,DELTA,RESULT=THETA
```

The instruction compares SIGMA and DELTA and places the resulting field in THETA. SIGMA and DELTA do not change.

Unless you specify otherwise, EDL assumes that the bit strings you specify are one-word (2-byte) variables. To compare a byte or more than two bytes, specify the number of consecutive units (bytes, words, or doublewords) that you want to compare. For example:

```
            EOR    CAIN,ABEL,(3,BYTE),RESULT=SETH
            .
            .
    CAIN    DATA   X'12A4E6'
    ABEL    DATA   X'0101'
    SETH    DATA   X'000000'
```

The instruction compares three bytes at CAIN with ABEL and places the result in SETH.

The Inclusive-OR Instruction

The inclusive-OR instruction (IOR) compares two bit strings and produces a third bit string, called the resulting field.

The instruction compares the two bit strings one bit at a time. If either or both bits are 1, the instruction sets a bit in the resulting field to 1. If neither bit is 1, the instruction sets a bit in the resulting field to 0.

The following example compares ETA to RHO and places the result in ETA.

```
IOR    ETA,RHO
```

The following table shows ETA and RHO before and after the instruction executes.

Data Item	Hex	Binary
ETA (before)	049C	0000 0100 1001 1100
RHO	56AB	0101 0110 1010 1011
ETA (after)	56BF	0101 0110 1011 1111

To compare a variable to a constant, code *operand2* as follows:

```
IOR    XI,X'5280'
```

The following table shows XI before and after the instruction executes.

Data Item	Hex	Binary
XI (before)	F0F0	1111 0000 1111 0000
constant	5280	0101 0010 1000 0000
XI (after)	F2F0	1111 0010 1111 0000

To compare two bit strings without altering the first operand, use the RESULT operand as follows:

```
IOR    PETER,PAUL,RESULT=MARY
```

The instruction compares PETER and PAUL and places the resulting field in MARY. PETER and PAUL do not change.

Unless you specify otherwise, EDL assumes that the bit strings you specify are one-word (2-byte) variables. To compare a byte or more than two bytes, specify the number of consecutive units (bytes, words, or doublewords) that you want to compare. For example:

```
IOR    PIG,COW,(4,DWORD),RESULT=POW
```

The instruction compares the first doubleword at PIG with the four doublewords at COW and places the resulting field in POW.

The AND Instruction

The AND instruction (AND) compares two bit strings and produces a third bit string, called the resulting field.

The instruction compares the two bit strings one bit at a time. If both bits are 1, the instruction sets a bit in the resulting field to 1. If either or both bits are 0, the instruction sets a bit in the resulting field to 0.

The following example compares BETA to THETA and places the result in BETA.

```
AND    BETA,THETA
```

The following table shows BETA both before and after the instruction executes.

Data Item	Hex	Binary
BETA (before)	049C	0000 0100 1001 1100
THETA	56AB	0101 0110 1010 1011
BETA (after)	0488	0000 0100 1000 1000

To compare a variable to a constant, code *operand2* as follows:

```
AND    LAMBDA,X'5280'
```

The following table shows LAMBDA both before and after the instruction executes.

Data Item	Hex	Binary
LAMBDA (before)	F0F0	1111 0000 1111 0000
constant	5280	0101 0010 1000 0000
LAMBDA (after)	5080	0101 0000 1000 0000

To compare two bit strings without altering the first operand, use the RESULT operand as follows:

```
AND    CEMENT,STONE,RESULT=WALL
```

The instruction compares CEMENT and STONE and places the resulting field in WALL. CEMENT and STONE do not change.

Unless you specify otherwise, EDL assumes that the bit strings you specify are one-word (2-byte) variables. To compare a byte or more than two words, specify the number of consecutive units (bytes, words, or doublewords) that you want to compare. For example:

```
AND    WALL,CEILING,(2,WORD),RESULT=ROOM
```

The instruction compares the first word at CEILING with the two words at WALL and places the resulting field in ROOM.

Writing Data from a Data Area

When you write data from a data area, the instruction you use depends on the kind of data and where you write it.

To write data to disk, diskette, or tape, use the WRITE instruction. To write data to a terminal, use either the PRINTEXT or PRINTNUM instruction. If the data is alphameric, use PRINTEXT. If the data consists of either one floating-point number or one or more integers, use PRINTNUM.

Writing Data to Disk or Diskette

You can write disk or diskette data sets either sequentially or directly. When you write, you always write 256 bytes, an "EDX record."

The following WRITE instruction writes a record sequentially:

```
          WRITE   DS3,DISKBUFF,1,0,ERROR=WRITERR
          .
          .
DISKBUFF  BUFFER  256,BYTES
```

The instruction writes a record to the third data set defined on the PROGRAM statement (DS3). The system writes one record (indicated by 1 in the third operand) sequentially (indicated by 0 in the fourth operand) into DISKBUFF. If an I/O error occurs, the program branches to WRITERR. Otherwise, the system writes the 256-byte buffer DISKBUFF to the data set.

The following WRITE instruction writes a record directly:

```
          WRITE   DS5,BUFR,1,RECNO,ERROR=BADWRIT
          .
          .
BUFR      BUFFER  256,BYTES
RECNO     DATA    F
```

The instruction writes a record to the fifth data set defined on the PROGRAM statement (DS5). The system writes one record (indicated by 1 in the third operand) directly (indicated by the presence of the label RECNO in the fourth operand) into BUFR. Where the system writes the record depends on the contents of RECNO. For example, if RECNO contains 150, the system writes the 150th record.

If an I/O error occurs, the program branches to BADWRIT. Otherwise, the system writes BUFR to the data set.

Writing Data to Tape

You can write tape data sets sequentially only. A tape WRITE writes a record from 18 to 32,767 bytes long.

The following WRITE instruction writes a record to a tape:

```
         WRITE    DS1,BUFF,1,327,ERROR=ERR,WAIT=YES
         .
         .
BUFF     BUFFER   327,BYTES
```

The system writes one record (indicated by 1 in the third operand). The size of the record is 327 bytes (indicated by 327 in the fourth operand). If an error occurs, control transfers to ERR. The system waits for the write operation to complete before continuing execution (WAIT=YES).

The buffer BUFF is 327 bytes long.

The following WRITE instruction writes 2 records from buffer BUFF2:

```
         WRITE    DS1,BUFF2,2,327,ERROR=ERR,WAIT=YES
         .
         .
BUFF2    BUFFER   768,BYTES
```

The system writes two records (indicated by 2 in the third operand). The size of each record is 327 bytes (indicated by 327 in the fourth operand). If an error occurs, control transfers to ERR. The system waits for the operation to complete before continuing (WAIT=YES). BUFF2 is 768 bytes long because it must be a multiple of 256.

Writing to a Terminal

Two of the instructions that write data to a terminal are the PRINTEXT and PRINTNUM instructions. The PRINTEXT instruction allows you to write alphameric data (alphabetic characters, numbers, and special characters). With the PRINTNUM instruction, you can write numbers (both integer and floating-point) only.

Writing Alphameric Data

To write alphameric data to a terminal, use the PRINTEXT instruction as follows:

```
         PRINTEXT DESC,SKIP=3
         .
         .
DESC     TEXT     'NOW IS THE TIME FOR ALL GOOD MEN'
```

The instruction writes (or *displays*) the 25 alphameric characters in DESC. The operand SKIP=3 causes the system to skip three lines before displaying DESC.

Unless you know how the system constructs a storage area defined by a TEXT statement, you should write from an area defined by a TEXT statement.

For information on writing alphameric data to screens, see Chapter 8, "Reading and Writing Data from Screens" on page PG-115.

Writing Data from a Data Area *(continued)*

Writing Numeric Data

The PRINTNUM instruction allows you to write either a single floating-point value or more than one integer to a terminal. The following instruction writes a floating-point number:

```
PRINTNUM   BASAL,TYPE=F,FORMAT=(6,2,F)
```

The instruction writes the number contained in the variable BASAL. The operand TYPE=F means that BASAL is a single-precision floating-point number. The operand FORMAT=(6,2,F) tells the system to display the number in 6 positions on the screen (including the decimal point), to display 2 digits to the right of the decimal point, and to display it as an "F-type" number such as 436.32.

To write more than one integer, code a second operand on the instruction as follows:

```
PRINTNUM WEIGHTS,7
```

The instruction displays the 7 one-word values starting at location WEIGHTS.

The instruction assumes that you have defined WEIGHTS as follows:

```
WEIGHTS   DATA   7F'0'
```

Controlling Program Logic

This section discusses the EDL instructions used to control the logic or execution of instructions. The following instructions are the primary means of controlling program logic:

- DO - initializes a loop
- ENDDO - ends a loop
- IF - tests a condition
- ELSE - specifies the action for a false condition
- ENDIF - ends an IF-ELSE structure
- GOTO - branches to another location

Relational Operators

The IF and DO statements involve the use of the following relational operators:

- EQ -- equal

- NE -- not equal
- GT -- greater than
- LT -- less than
- GE -- greater than or equal
- LE -- less than or equal

The IF Instruction

The IF instruction allows you to compare two areas of storage. 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'FFFD', however, as a -3. Though X'FFFD' seems to be a larger hexadecimal number than X'0FFF', the system recognizes X'FFFD' as a negative number and X'0FFF' as a positive number. X'FFFD' is a negative number to the system because the leftmost bit is "on".

When you compare data logically, the system compares the data byte-by-byte. The system interprets X'FFFF' as 2 bytes with all bits "on".

Comparing Data Arithmetically

The form of the arithmetic comparison is:

```
IF (data1,operator,data2,width)
```

If *data1* has the relationship indicated by *operator* to *data2*, the next sequential instruction executes. *Width* indicates the length of the data to be compared and must be BYTE, WORD (the default), DWORD, FLOAT, or DFLOAT.

This is called the true portion of the IF-ELSE-ENDIF structure. For example:

```
IF    (A,EQ,B,WORD)
  PRINTNUM  A
ELSE
  PRINTNUM  B
ENDIF
```

ELSE is an optional part of the structure. The instructions following it are called the false part of the structure. Therefore, in the preceding example, the instruction following the ELSE instruction executes if A is not equal to B. If ELSE is not coded and the condition is false, control passes to the instruction following the ENDIF.

You can test more than two conditions in a single IF statement.

```
IF    (ALPHA,LT,BETA),AND,(GAMMA,NE,DELTA)
```

IF ALPHA is less than BETA *and* GAMMA is not equal to DELTA, the next sequential instruction executes.

You can also execute the next sequential instruction if either test produces a true condition.

```
IF    (PI,GE,PSI),OR,(CHI,NE,OMEGA)
```

If PI is greater than or equal to PSI *or* CHI is not equal to OMEGA, the next sequential instruction executes.

To compare a variable to a constant, code the constant as *data2* as follows:

```
IF    (FEET,EQ,5280)
```

If FEET equals 5280 (decimal), the next sequential instruction executes.

Comparing Data Logically

The form of the logical comparison is:

```
IF (data1,operator,data2,width)
```

If *data1* has the relationship indicated by *operator* to *data2*, the next sequential instruction executes. *Width* indicates the length of the data to be compared and must be an integer.

For example:

```
IF    (A,GE,B,4)
  PRINTNUM  A
ELSE
  PRINTNUM  B
ENDIF
```

The instruction(s) that follow the IF instruction is (are) called the true portion of the IF-ELSE-ENDIF structure. If the 4 bytes in A are greater than or equal to the 4 bytes in B, the next sequential instruction executes.

The instruction(s) following the ELSE instruction is (are) called the false part of the structure. ELSE is an optional part of the structure. If the 4 bytes in A are *not* greater than or equal to the 4 bytes in B, the instruction following the ELSE instruction executes.

If the ELSE instruction is not coded and the condition is false, control passes to the instruction following the ENDIF.

The Program Loop

The DO instruction allows you to execute the same code repetitively. The DO instruction starts a DO loop and the ENDDO instruction ends the loop. The loop consists of the instructions between the DO and ENDDO. The following sections show the different forms of the DO loop.

The Simple DO

The loop executes a specified number of times.

```
DO   100,TIMES
  GETVALUE  PSI,PROMPT3
  ADD       COUNT,PSI
ENDDO
```

The GETVALUE and ADD instruction execute 100 times.

The DO UNTIL

The loop executes until the condition occurs. (The loop always executes at least once.)

```
DO    UNTIL,(CDED,GT,1000,FLOAT)
  GETVALUE  OMICRON,OMPRMPT
  FSUB      CDED,OMICRON
ENDDO
```

The GETVALUE and FSUB instructions execute until CDED is greater than 1000.

The DO WHILE

The loop executes as long as the condition exists.

```
DO    WHILE,(B,NE,C)
  GETVALUE B,'ENTER B'
  GETVALUE C,'ENTER C'
ENDDO
```

The GETVALUE instructions execute as long as B does not equal C.

The Nested DO Loop

A DO loop can contain other DO loops. For example:

```
DO    UNTIL,(ALPHA,LT,BETA,DFLOAT),OR,(#1,EQ,1000)
  GETVALUE ALPHA,'ENTER ALPHA',TYPE=L,FORMAT=(12,3,E)
  GETVALUE BETA,'ENTER BETA',TYPE=L,FORMAT=(12,3,E)
  MOVE     #1,BETA,(1,DFLOAT)
  DO  10,TIMES
    FADD  GAMMA,ALPHA,PREC=LLL
  ENDDO
ENDDO
```

The FADD statement contained in the inner DO executes 10 times for each execution of the outer DO.

The Nested IF Instruction

A DO loop can also contain IF statements. For example:

```
READTEXT  CHAR,'ENTER A CHARACTER'
GETVALUE  A,'ENTER A'
GETVALUE  B,'ENTER B'
DO  WHILE,(A,GT,B)
  IF  (CHAR,EQ,C'A',BYTE)
    DO  40,TIMES
      .
      .
      .
    ENDDO
  ELSE
      .
      .
      .
  ENDIF
  GETVALUE  A,'ENTER A'
  GETVALUE  B,'ENTER B'
ENDDO
```

The outer DO loop executes as long as A is greater than B. The inner DO loop executes 40 times if CHAR equals the letter A.

Controlling Program Logic *(continued)*

Branching to Another Location

The GOTO instruction allows you to transfer control to another location within a program. For example, the following instruction transfers control to the instruction at label LOC1:

```
    GOTO   LOC1
```

To branch to an address defined by a label, enclose the label in parentheses as follows:

```
    GOTO   (CALC)
```

This instruction branches to the address contained in CALC. You must define CALC as an address variable as in the following DATA statement:

```
CALC     DATA   A(RTN01)
```

To branch to a location that is based on the contents of a variable, code the GOTO statement like this:

```
    GOTO   (ERR,L1,L2),I
```

The instruction branches to L1 if I equals 1, to L2 if I equals 2, and to ERR for any other value of I. The system branches to the first label in parentheses if the variable is less than 1 or greater than the number of labels minus 1.

Referring to a Storage (Program) Location

You can use the EQU statement to refer to the next available storage location in a program. You can use it to generate labels in your program. For example:

```
  CALLA    EQU     *
           MOVE    C,+A,(1,BYTE)
           .
           .
           .
           GOTO    CALLA
```

Ending the Program

Ending a program requires three statements: PROGSTOP, ENDPROG, and END.

The PROGSTOP statement ends the program and releases any storage that it used. It also signals the end of the executable instructions.

The ENDPROG statement follows the statements that define storage areas and precedes the END statement.

The END statement follows the ENDPROG statement. It tells the compiler that the program contains no more statements.

The following example shows the position of the three statements and the general structure of a program.

```
PRINT     PROGRAM   START
START     EQU       *
          .
          .
          .
          PROGSTOP
FIELD1    DATA      F'0'
          .
          .
          ENDPROG
          END
```

Notes

Chapter 3. Entering a Source Program

After you code a source program, you must enter it into a data set. The data set can be on either disk, diskette, or tape.

This chapter shows how to use the text editor called the $FSEDIT utility. The chapter describes the commands you need to enter a new source program or change an existing source program. For a complete list of $FSEDIT commands, refer to *Operator Commands and Utilities Reference*.

Invoking the Editor

You can invoke the editor in one of two ways. You can load it directly using the $L command. Or, you can invoke it using the session manager.

This chapter discusses how to invoke the editor with the session manager. For information on how to invoke $FSEDIT with the $L command, refer to *Operator Commands and Utilities Reference*.

As you learned in Chapter 1 of this book, you load the session manager by pressing the attention key, typing **$L $SSMAIN**, and pressing the enter key.

At this point, enter a one to four character ID and press the enter key.

The Session Manager Primary Option Menu appears. From this menu, select option 1 (TEXT EDITING). The session manager displays the $FSEDIT Primary Option Menu.

Invoking the Editor *(continued)*

Creating a New Data Set

The session manager allocates data sets automatically when you log on. One of these data sets, a work data set used by $FSEDIT, is named $SMExxxx, where **xxxx** is the ID you entered when you logged on to the session manager. For example, if you entered **ABCD** when you logged on, the work data set is $SMEABCD.

Use option 2 (EDIT) to put your source program into the work data set.

```
$FSEDIT PRIMARY OPTION MENU  --------------------------------STATUS = INIT
                                                            PRESS PF3 TO EXIT
OPTION ===> 2

DATASET NAME =========>          (CURRENTLY IN WORK DATASET)
VOLUME NAME ==========>

HOST DATASET ========>

ENTER A VOLUME NAME AND PRESS ENTER FOR A DIRECTORY LIST.

1 ---- BROWSE
2 ---- EDIT
3 ---- READ (HOST/NATIVE)
4 ---- WRITE (HOST/NATIVE)
5 ---- SUBMIT BATCH JOB TO HOST SYSTEM
6 ---- PRINT
7 ---- MERGE
8 ---- END
9 ---- HELP
```

An empty data set appears on your screen. The name of the data set and the volume on which it resides are shown at the top of the screen.

```
EDIT --- $SMEABCD, EDX003     0(1089)------------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ***********************************************************
.....
***** ***** BOTTOM OF DATA ********************************************************
```

The cursor is located at the first input line. After you finish typing text on this line, press the enter key.

The following example shows how the screen looks after you enter the first line of a source program. (We have used the source program described in Chapter 1 of this book.) The editor automatically numbers each line and presents a new blank line.

```
EDIT --- $SMEABCD, EDX003      0(1089)------------------------- COLUMNS 001 072
COMMAND INPUT ===>                                              SCROLL ==> HALF
***** ***** TOP OF DATA *******************************************************
00010 ADD10       PROGRAM   STPGM
.....
***** ***** BOTTOM OF DATA ****************************************************
```

Continue to type each line of your source program. When you finish, press the enter key on a blank line.

```
EDIT --- $SMEABCD , EDX003    12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ******************************************************
+00010 ADD10       PROGRAM      STPGM
+00020 STPGM       GETVALUE     COUNT,'ENTER NUMBER: '
+00030 LOOP        DO           10,TIMES
+00040               ADD        SUM,COUNT
+00050             ENDDO
+00060             PRINTEXT     '@RESULT='
+00070             PRINTNUM     SUM
+00080             PROGSTOP
+00090 COUNT       DATA         F'0'
+00100 SUM         DATA         F'0'
+00110             ENDPROG
+00120             END
***** ***** BOTTOM OF DATA ***************************************************
```

Saving Your Data Set

The next step is to save your data set. Return to the $FSEDIT Primary Option Menu by typing M (for "menu") on the COMMAND INPUT line.

Select option 4 (WRITE) to save the data set. Type the name next on the DATASET NAME line. (In this example, we named the data set ADD10. Type the volume on the VOLUME NAME line. (In this example, the volume is EDX002.) Then press the enter key.

```
$FSEDIT PRIMARY OPTION MENU ----------------------------STATUS = MODIFIED
                                                        PRESS PF3 TO EXIT
OPTION ===> 4

DATASET NAME =========> ADD10      (CURRENTLY IN WORK FILE)
VOLUME NAME ==========> EDX002

HOST DATASET ========>

ENTER A VOLUME NAME AND PRESS ENTER FOR A DIRECTORY LIST.

1 ---- BROWSE
2 ---- EDIT
3 ---- READ (HOST/NATIVE)
4 ---- WRITE (HOST/NATIVE)
5 ---- SUBMIT
6 ---- PRINT
7 ---- MERGE
8 ---- END
9 ---- HELP
```

Next, the system prompts you as follows:

```
WRITE TO ADD10 ON EDX002 (Y/N)?
```

Type Y and press the enter key.

Then you see a message on your screen indicating that the data set has been written to the volume. In the example shown above, the following message would appear:

```
12 LINES WRITTEN TO ADD10,EDX002
```

This message means that the source program is 12 records long and has been written to volume EDX002.

Modifying an Existing Data Set

You have seen how to enter a source program into a new data set. You can also modify an existing data set.

You must first read the data set you want to modify into the work data set. Select option 3 (READ) from the $FSEDIT Primary Options Menu. On the menu, you specify which data set you want to read.

Next, you select option 2 (EDIT) to modify the data set.

The data set appears on your screen.

```
EDIT --- ADD10     , EDX002    12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                              SCROLL ==> HALF
***** ***** TOP OF DATA ******************************************************
00010 ADD10          PROGRAM        STPGM
00020 STPGM          GETVALUE       COUNT,'ENTER NUMBER: '
00030 LOOP           DO             10,TIMES
00040                  ADD          SUM,COUNT
00050                ENDDO
00060                PRINTEXT       '@RESULT='
00070                PRINTNUM       SUM
00080                PROGSTOP
00090 COUNT          DATA           F'0'
00100 SUM            DATA           F'0'
00110                ENDPROG
00120                END
***** ***** BOTTOM OF DATA ***************************************************
```

Changing a Line

To change a line, move the cursor to the line and type in the correction. For example, suppose you wanted to change **10** to **15** in the DO instruction. Move the cursor to the **0** and type a **5**.

Or, suppose you wanted to delete the = character in the PRINTEXT instruction. You would move the cursor to the = character and press the delete key.

Modifying an Existing Data Set *(continued)*

Inserting a Line

You can insert a new line into your data set. You insert a line by typing an I in the line number *after which* you want to insert.

For example, suppose you want to insert another instruction before PROGSTOP. Type the I as follows:

```
EDIT --- ADD10    , EDX002    12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===> _                                           SCROLL ==> HALF
***** ***** TOP OF DATA ******************************************************
00010 ADD10         PROGRAM       STPGM
00020 STPGM         GETVALUE      COUNT,'ENTER NUMBER: '
00030 LOOP          DO            10,TIMES
00040                 ADD         SUM,COUNT
00050               ENDDO
00060               PRINTEXT      '@RESULT='
I0070               PRINTNUM      SUM
00080               PROGSTOP
00090 COUNT         DATA          F'0'
00100 SUM           DATA          F'0'
00110               ENDPROG
00120               END
***** ***** BOTTOM OF DATA ***************************************************
```

After you press the enter key, your data set looks like this:

```
EDIT --- ADD10    , EDX002    12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ******************************************************
00010 ADD10         PROGRAM       STPGM
00020 STPGM         GETVALUE      COUNT,'ENTER NUMBER: '
00030 LOOP          DO            10,TIMES
00040                 ADD         SUM,COUNT
00050               ENDDO
00060               PRINTEXT      '@RESULT='
00070               PRINTNUM      SUM
.....               _
00080               PROGSTOP
00090 COUNT         DATA          F'0'
00100 SUM           DATA          F'0'
00110               ENDPROG
00120               END
***** ***** BOTTOM OF DATA ***************************************************
```

You could now enter your new line of text at the position of the cursor. After you press enter, the editor assigns a line number to your new line of text. A new blank input line also appears. You can continue to insert lines or you can press the enter key again to indicate that you have finished inserting.

Modifying an Existing Data Set *(continued)*

Deleting a Line

You can delete a line or series of lines from your data set.

To delete a single line, enter a **D** in the line number you want deleted and press the enter key.

```
EDIT --- ADD10    , EDX002     13( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                              SCROLL ==> HALF
***** ***** TOP OF DATA *******************************************************
00010 ADD10        PROGRAM        STPGM
00020 STPGM        GETVALUE       COUNT,'ENTER NUMBER: '
00030 LOOP         DO             10,TIMES
00040                ADD          SUM,COUNT
00050              ENDDO
00060              PRINTEXT       '@RESULT='
00070              PRINTNUM       SUM
D0080 *********Delete this line*********************
00090              PROGSTOP
00100 COUNT        DATA           F'0'
00110 SUM          DATA           F'0'
00120              ENDPROG
00130              END
***** ***** BOTTOM OF DATA ****************************************************
```

After you press the enter key, the editor deletes the line.

```
EDIT --- ADD10    , EDX002     12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                              SCROLL ==> HALF
***** ***** TOP OF DATA *******************************************************
00010 ADD10        PROGRAM        STPGM
00020 STPGM        GETVALUE       COUNT,'ENTER NUMBER: '
00030 LOOP         DO             10,TIMES
00040                ADD          SUM,COUNT
00050              ENDDO
00060              PRINTEXT       '@RESULT='
00070              PRINTNUM       SUM
00090              PROGSTOP
00100 COUNT        DATA           F'0'
00110 SUM          DATA           F'0'
00120              ENDPROG
00130              END
***** ***** BOTTOM OF DATA ****************************************************
```

Modifying an Existing Data Set *(continued)*

You can also delete more than one lines.

For example, suppose you want to delete lines 80 through 120 in the following program. Type **DD** in line 80 and another **DD** in line 120.

```
EDIT --- ADD10    , EDX002    17( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ***********************************************************
00010 ADD10         PROGRAM        STPGM
00020 STPGM         GETVALUE       COUNT,'ENTER NUMBER: '
00030 LOOP          DO             10,TIMES
00040                 ADD          SUM,COUNT
00050               ENDDO
00060               PRINTEXT       '@RESULT='
00070               PRINTNUM       SUM
DD080 **********Delete these lines*********************
00090 ******                                    ******
00100 ******                                    ******
00110 ******                                    ******
DD120 **********Delete these lines*********************
00130               PROGSTOP
00140 COUNT         DATA           F'0'
00150 SUM           DATA           F'0'
00160               ENDPROG
00170               END
***** ***** BOTTOM OF DATA ****************************************************
```

After you press the enter key, your program looks like this:

```
EDIT --- ADD10    , EDX002    12( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ***********************************************************
00010 ADD10         PROGRAM        STPGM
00020 STPGM         GETVALUE       COUNT,'ENTER NUMBER: '
00030 LOOP          DO             10,TIMES
00040                 ADD          SUM,COUNT
00050               ENDDO
00060               PRINTEXT       '@RESULT='
00070               PRINTNUM       SUM
00130               PROGSTOP
00140 COUNT         DATA           F'0'
00150 SUM           DATA           F'0'
00160               ENDPROG
00170               END
***** ***** BOTTOM OF DATA ****************************************************
```

The editor deletes the lines.

Moving Lines

You can move a line or series of lines from one part of your data set to another.

For example, suppose you want to move lines 110 through 130. First type MM in both 110 and 130:

If you want to move these lines after line 10, place an A (for "after") on line 10 and press the enter key.

```
EDIT --- ADD10    , EDX002    15( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ***************************************************
A0010 ADD10         PROGRAM       STPGM
00020 STPGM         GETVALUE      COUNT,'ENTER NUMBER: '
00030 LOOP          DO            10,TIMES
00040                 ADD         SUM,COUNT
00050               ENDDO
00060               PRINTEXT      '@RESULT='
00070               PRINTNUM      SUM
00080               PROGSTOP
00090 COUNT         DATA          F'0'
00100 SUM           DATA          F'0'
MM110 *********Move these lines***********************
00120 *****                                      *****
MM130 *********Move these lines***********************
00140               ENDPROG
00150               END
***** ***** BOTTOM OF DATA ************************************************
```

When you press the enter key, the editor moves the lines to the position *after* line 10.

```
EDIT --- ADD10    , EDX002    15( 1089)----------------------- COLUMNS 001 072
COMMAND INPUT ===>                                             SCROLL ==> HALF
***** ***** TOP OF DATA ***************************************************
00010 ADD10         PROGRAM       STPGM
+00020 *********Move these lines***********************
+00030 *****                                      *****
+00040 *********Move these lines***********************
00050 STPGM         GETVALUE      COUNT,'ENTER NUMBER: '
00060 LOOP          DO            10,TIMES
00070                 ADD         SUM,COUNT
00080               ENDDO
00090               PRINTEXT      '@RESULT='
00100               PRINTNUM      SUM
00110               PROGSTOP
00120 COUNT         DATA          F'0'
00130 SUM           DATA          F'0'
00140               ENDPROG
00150               END
***** ***** BOTTOM OF DATA ************************************************
```

Entering a Source Program

Modifying an Existing Data Set *(continued)*

After you make changes to your data set, return to the $FSEDIT Primary Options Menu. Return to that menu by typing M (for "menu") on the COMMAND INPUT line. To save the changes, select option 4 and press the enter key.

You have seen how you can change lines in your programs. You have also seen how to insert and delete lines and move a series of lines. The session manager was used to invoke $FSEDIT and to allocate the necessary data sets.

The next chapter explains how to compile your programs using $EDXASM, the EDX compiler.

Chapter 4. Compiling a Program

After you design, code, and enter your source program into a data set, you have to compile the source program into an object module. This chapter shows you how to compile your source program using the Event Driven Language Compiler, $EDXASM.

The chapter also shows a step-by-step example of compiling a source program that contains some syntax errors. The chapter then shows how to correct the errors so that the compilation is successful.

You can invoke $EDXASM in one of three ways. You can load $EDXASM directly using the $L command. You can use the $JOBUTIL utility to invoke $EDXASM. Or, you can run your compilation under control of the session manager.

This chapter describes how to compile a program using the session manager.

For information on using the $L command or the $JOBUTIL utility, see *Operator Commands and Utilities Reference*.

Allocating Data Sets

When you use $EDXASM under control of the session manager, you must provide two data sets. The first data set is the actual source program to be compiled. You must have entered the source program on a disk, diskette, or tape data set. Chapter 3, "Entering a Source Program" on page PG-59 describes how to use the $FSEDIT utility to enter your source programs.

The output of the compiler is a data set that contains an object module. You can allocate this data set by selecting option 3 (DATA MANAGEMENT) from the Session Manager Primary Option Menu.

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU -----------------------------
ENTER/SELECT PARAMETERS:                                   PRESS PF3 TO EXIT

                                          19:42:07
      SELECT OPTION ==> 3                 10/24/82
                                          ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

Note: This example assumes that you logged on to the Session Manager with an ID of ABCD.

The Data Management Option Menu appears on the screen. To allocate your object code data set, select option 1 ($DISKUT1).

```
$SMM03  SESSION MANAGER DATA MANAGEMENT OPTION MENU-----------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

  SELECT OPTION ==> 1

   1 - $DISKUT1 (DISK(ETTE) ALLOCATE, LIST DIRECTORY)
   2 - $DISKUT2 (DISK(ETTE) DUMP/LIST DATASETS)
   3 - $COPYUT1 (DISK(ETTE) COPY DATASETS/VOLUMES)
   4 - $COMPRES (DISK(ETTE) COMPRESS A VOLUME)
   5 - $COPY    (DISK(ETTE) COPY DATASETS/VOLUMES)
   6 - $DASDI   (DISK(ETTE) SURFACE INITIALIZATION)
   7 - $INITDSK (DISK(ETTE) INITIALIZE/VERIFY)
   8 - $MOVEVOL (COPY DISK VOLUME TO MULTI-DISKETTES)
   9 - $IAMUT1  (INDEXED ACCESS METHOD UTILITY PROGRAM)
  10 - $TAPEUT1 (TAPE ALLOCATE, CHANGE, COPY)
  11 - $HXUT1   (H-EXCHANGE DATASET UTILITY)

   WHEN ENTERING THESE UTILITIES, THE USER IS EXPECTED
   TO ENTER A COMMAND. IF A QUESTION MARK (?) IS ENTERED
   INSTEAD OF A COMMAND, THE USER WILL BE PRESENTED WITH
   A LIST OF AVAILABLE COMMANDS.
```

The session manager loads the $DISKUT1 utility and prompts for the command you want to use.

```
> $L $DISKUT1
LOADING $DISKUT1      59P,19:44:28, LP= 9200, PART=1

$DISKUT1 - DATA SET MANAGEMENT UTILITY I

USING VOLUME EDX002

COMMAND (?): _
```

Notice the USING VOLUME EDX002 message. Unless you change volumes, $DISKUT1 allocates your data set on EDX002.

To change the default volume, enter a **CV** command.

To change the default volume to MYVOL, enter the following CV command:

```
USING VOLUME EDX002

COMMAND (?): CV MYVOL
```

The system responds with:

```
USING VOLUME MYVOL

COMMAND (?): _
```

Use the CV command only when you do *not* want to use the default volume.

Use the **AL** command to allocate your data set.

```
COMMAND (?): AL
MEMBER NAME:
```

The system then prompts you for the name of the data set. In this example, the data set name is OBJECT.

```
MEMBER NAME: OBJECT
HOW MANY RECORDS? _
```

Next, the system prompts for the number of records you want to allocate. A 25- to 50-record data set should be large enough for most programs. This example defines a 25-record data set.

```
HOW MANY RECORDS? 25
DEFAULT TYPE = DATA - OK(Y/N)? _
```

Finally, the system prompts for the type of information to be contained in the data set. The default is DATA. Because this data set will contain data, enter a Y.

```
DEFAULT TYPE = DATA - OK(Y/N)? Y
```

The system responds with:

```
OBJECT CREATED

COMMAND (?):
```

Once the data set has been created, enter an EN (for "end") to return to the Data Management Option Menu screen.

```
COMMAND (?): EN

$DISKUT1 ENDED 08:30:24
```

Return to the Session Manager Primary Option Menu to begin the compilation by pressing the PF3 key.

Running the Compilation

Once you have allocated the data set to hold the output, you are ready to begin compiling the source program. The following is a listing of the source program to be compiled:

```
          PROGRAM      STPGM
STPGM     GETVALUE     COUNT,'ENTER NUMBER: '
LOOP      DO           10,TIMES
            ADD        SUM,COUNT
          ENDDO
          PRINTEXT     'RESULT='
          PRINTNUM     SUM
          PROGSTOP
COUNT     DATA         F'0'
SUM       DATA         F'0'
          ENDPROG
          END
```

This program is similar to the examples we used in Chapter 1 and Chapter 3 of this book. However, we have included two errors in this source program.

From the Session Manager Primary Option Menu, select option 2 (PROGRAM PREPARATION) to begin the compile step.

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ----------------------------
ENTER/SELECT PARAMETERS:                                 PRESS PF3 TO EXIT

                                            19:48:07
       SELECT OPTION ==> 2                  10/24/82
                                            ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARATION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

The Program Preparation Option Menu appears on your screen. To compile the program, select option 1 ($EDXASM COMPILER).

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU-------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

       SELECT OPTION ==> 1

            1 - $EDXASM COMPILER
            2 - $EDXASM/$EDXLINK
            3 - $S1ASM ASSEMBLER
            4 - $COBOL COMPILER
            5 - $FORT FORTRAN COMPILER
            6 - $PLI COMPILER/$EDXLINK
            7 - $EDXLINK LINKAGE EDITOR
            8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISOR
            9 - $UPDATE
           10 - $UPDATEH (HOST)
           11 - $PREFIND
           12 - $PASCAL COMPILER/$EDXLINK
           13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

The $EDXASM Parameter Input Menu appears on your screen. Enter the name of your source input (in this example, ADD10 on volume EDX002). Also enter the name of your object output (in this example, data set OBJECT on volume MYVOL).

You could enter something on the OPTIONAL PARAMETERS line if you want to change one of the parameters listed on the DEFAULT PARAMETERS line. In this example, we are using the defaults.

```
$SMM0201: SESSION MANAGER $EDXASM PARAMETER INPUT MENU--------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

  SOURCE INPUT   (NAME,VOLUME) ==> ADD10,EDX002

  OBJECT OUTPUT  (NAME,VOLUME) ==> OBJECT,MYVOL

  OPTIONAL PARAMETERS ==>
  (SELECT FROM THE LIST BELOW)

  -------------------------------------------------------
AVAILABLE PARAMETERS:     ABBREVIATION:          DESCRIPTION:
 NOLIST                    NO                     USED TO SUPPRESS LISTING
 LIST TERMINAL-NAME        LI TERMINAL-NAME       USE LIST * FOR THIS TERMINAL
 ERRORS TERMINAL-NAME      ER TERMINAL-NAME       USE ERRORS * FOR THIS TERMINAL
 CONTROL DATA SET,VOLUME   CO DATA SET,VOLUME     $EDXASM LANGUAGE CONTROL DATASET
 OVERLAY #                 OV #                   # IS NUMBER OF AREAS FROM 1 TO 6

DEFAULT PARAMETERS:
 LIST $SYSPRTR CONTROL $EDXL,ASMLIB OVERLAY 4
```

Checking Your Compiler Listing and Correcting Errors

The output of the compiler prints on your printer. The listing consists of statistics, source code statements and object code, undefined or external symbols, and a completion code.

The following is an example of the output listing generated by the compile example being run.

```
EDX ASSEMBLER STATISTICS

 SOURCE INPUT   - ADD10,EDX002
 WORK DATA SET  - WORK1,MYVOL
 OBJECT MODULE  - OBJECT,MYVOL
 DATE: 10/24/82 AT 19:56:18
 ASSEMBLY TIME:      4 SECONDS
 STATEMENTS PROCESSED -  12

    4  STATEMENTS FLAGGED
                                                                         PAGE   1
LOC     +0    +2    +4    +6    +8    SOURCE STATEMENT   ADD10     ,EDX002  (5719

                                            PROGRAM      STPGM
   08 *** TASK NAME NOT SPECIFIED                                      $EDXL   12
0000   802C 0000 000A 0001 0E0E   STPGM    GETVALUE     COUNT,'ENTER NUMBER: '
000A   C5D5 E3C5 D940 D5E4 D4C2
0014   C5D5 7A40
   08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED                 $EDXL    3
0018   809C 0024 000A             LOOP     DO           10,TIMES
001E   0032 0040 0000                      ADD          SUM,COUNT
   08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED                 $EDXL    3
0024   009D 0000 0001                      ENDDO
002A   8026 0808 D9C5 E2E4 D3E3            PRINTEXT       'RESULT='
0034   7E40                                PRINTNUM       SUM
003C   0022 FFFF                           PROGSTOP
                                  COUNT    DATA           F'0'
   08 *** INVALID OR UNDEFINED OPERATION CODE                          $EDXL   11
0040   0000                       SUM      DATA           F'0'
0042                                       ENDPROG
0042                                       END


                          EXTERNAL/UNDEFINED SYMBOLS

                             COUNT        UNDEFINED

COMPLETION CODE =      8
```

The previous example shows that the compile did not run successfully. The completion code expected is a -1. The completion code received is an 8.

The listing shows the compilation errors. They are:

- 08 *** TASK NAME NOT SPECIFIED
- 08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED
- 08 *** INVALID OR UNDEFINED OPERATION CODE

To fix these errors, you must understand what caused them. Look the errors up in *Messages and Codes*.

The first message, 08 *** TASK NAME NOT SPECIFIED, is a result of not having a taskname coded on the PROGRAM statement.

The second message, 08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED, means that one of the labels referenced in the instruction has not been defined to the program. If you check the listing for undefined symbols, you will see that COUNT is undefined.

The third message, 08 *** INVALID OR UNDEFINED OPERATION CODE, means that something is wrong with the COUNT definition statement. If you check the statement, you will see that the label, COUNT, starts in column two. The label must start in column one.

After isolating the errors, you must go back to the source data set and correct them. Use $FSEDIT as explained in Chapter 3, "Entering a Source Program" on page PG-59 to make the corrections. After you make the corrections, the source data set looks as follows:

```
PROG1         PROGRAM      STPGM
STPGM         GETVALUE     COUNT,'ENTER NUMBER: '
LOOP          DO           10,TIMES
                ADD        SUM,COUNT
              ENDDO
              PRINTEXT     '@RESULT='
              PRINTNUM     SUM
              PROGSTOP
COUNT         DATA         F'0'
SUM           DATA         F'0'
              ENDPROG
              END
```

Rerunning the Compilation

To rerun the compilation, return to the Session Manager Primary Option Menu.

From the Session Manager Primary Option Menu, select option 2 (PROGRAM PREPARATION).

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU -----------------------------
ENTER/SELECT PARAMETERS:                                  PRESS PF3 TO EXIT

                                          20:02:07
       SELECT OPTION ==> 2                 10/24/82
                                          ABCD

            1 - TEXT EDITING
            2 - PROGRAM PREPARA"ION
            3 - DATA MANAGEMENT
            4 - TERMINAL UTILITIES
            5 - GRAPHICS UTILITIES
            6 - EXEC PROGRAM/UTILITY
            7 - EXEC $JOBUTIL PROC
            8 - COMMUNICATION UTILITIES
            9 - DIAGNOSTIC AIDS
```

The Program Preparation Option Menu appears on your screen. Select option 1 ($EDXASM COMPILER).

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU-------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

       SELECT OPTION ==> 1

            1 - $EDXASM COMPILER
            2 - $EDXASM/$EDXLINK
            3 - $S1ASM ASSEMBLER
            4 - $COBOL COMPILER
            5 - $FORT FORTRAN COMPILER
            6 - $PLI COMPILER/$EDXLINK
            7 - $EDXLINK LINKAGE EDITOR
            8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISORS
            9 - $UPDATE
           10 - $UPDATEH (HOST)
           11 - $PREFIND
           12 - $PASCAL COMPILER/$EDXLINK
           13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

The $EDXASM Parameter Input Menu appears on your screen. Again, enter the name of your source input (in this example, ADD10). Also enter the name of your object output (in this example, data set OBJECT on volume MYVOL).

```
$SMM0201: SESSION MANAGER $EDXASM PARAMETER INPUT MENU--------------------
ENTER/SELECT PARAMETERS:                                 PRESS PF3 TO RETURN

  SOURCE INPUT   (NAME,VOLUME) ==> ADD10,EDX002

  OBJECT OUTPUT  (NAME,VOLUME) ==> OBJECT,MYVOL

  OPTIONAL PARAMETERS ==>
  (SELECT FROM THE LIST BELOW)

 --------------------------------------------------------
AVAILABLE PARAMETERS:      ABBREVIATION:          DESCRIPTION:
 NOLIST                     NO                     USED TO SUPPRESS LISTING
 LIST TERMINAL-NAME         LI TERMINAL-NAME       USE LIST * FOR THIS TERMINAL
 ERRORS TERMINAL-NAME       ER TERMINAL-NAME       USE ERRORS * FOR THIS TERMINAL
 CONTROL DATA SET,VOLUME    CO DATA SET,VOLUME     $EDXASM LANGUAGE CONTROL DATASET
 OVERLAY #                  OV #                   # IS NUMBER OF AREAS FROM 1 TO 6

DEFAULT PARAMETERS:
 LIST $SYSPRTR CONTROL $EDXL,ASMLIB OVERLAY 4
```

Rerunning the Compilation *(continued)*

The following is an example of the output listing generated by the compiler.

```
EDX ASSEMBLER STATISTICS

 SOURCE INPUT   - ADD10,EDX002
 WORK DATA SET - $SM1ABCD,EDX002
 OBJECT MODULE - OBJECT,MYVOL
 DATE: 10/24/82 AT 20:06:18
 ASSEMBLY TIME:     4 SECONDS
 STATEMENTS PROCESSED -  12

   NO  STATEMENTS FLAGGED

LOC     +0    +2    +4    +6    +8    SOURCE STATEMENT   ADD10     ,EDX002  (5719

0000    0008 D7D9 D6D7 D9C1 D440   PROG1     PROGRAM      STPGM

0034    802C 0074 003E 0001 0E0E   STPGM     GETVALUE     COUNT,'ENTER NUMBER: '
003E    C5D5 E3C5 D940 D5E4 D4C2
0048    C5D9 7A40
004C    809C 0058 000A             LOOP      DO           10,TIMES
0052    0032 0076 0074                       ADD          SUM,COUNT
0058    009D 0000 0001                       ENDDO
005E    8026 0808 D9C5 E2E4 D3E3             PRINTEXT       'RESULT='
0068    7E40
006A    0028 0076 0001                       PRINTNUM       SUM
0070    0022 FFFF                            PROGSTOP
0074    0000                       COUNT     DATA           F'0'
0076    0000                       SUM       DATA           F'0'
0078    0000 0000 0000 0234 0000             ENDPROG
             .
00FA    0000 0000 0000 0000 0000
010E    0000
0110                                             END

                        EXTERNAL/UNDEFINED SYMBOLS

                           SVC        WXTRN
                           SUPEXIT    WXTRN
                           SETBUSY    WXTRN

COMPLETION CODE =     -1
```

The -1 completion code tells you that the compile was successful. The next step is to link-edit the object module into program data that can be executed. See the next chapter, Chapter 5, "Preparing an Object Module for Execution" on page PG-81, for details.

Chapter 5. Preparing an Object Module for Execution

So far in this book, you have learned how to code and enter a source program into a data set. You have also learned how to compile the source program.

The next step is to prepare your object modules for execution. In this chapter, we will show you how to use the linkage editor $EDXLINK to prepare your object modules to run on an EDX system. $EDXLINK links together any separately assembled object modules that make up your program. $EDXLINK also produces a load module that is ready for execution.

In this chapter, we will show you how to prepare a single object module for execution. We will also show you an example of link-editing more than one object module.

You can invoke $EDXLINK in one of three ways. You can load $EDXLINK directly using the $L command. You can use the $JOBUTIL utility to invoke $EDXLINK. Or, you can use $EDXLINK under control of the session manager.

This chapter describes how to use $EDXLINK under control of the session manager. For information on using the $L command or the $JOBUTIL utility, refer to *Operator Commands and Utilities Reference.*

Link-Editing a Single Object Module

This section shows how to link-edit a single object module.

$EDXLINK LINKAGE EDITOR is option 7 of the Session Manager Program Preparation Option menu.

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU--------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

      SELECT OPTION ==> 7

           1 - $EDXASM COMPILER
           2 - $EDXASM/$EDXLINK
           3 - $S1ASM ASSEMBLER
           4 - $COBOL COMPILER
           5 - $FORT FORTRAN COMPILER
           6 - $PLI COMPILER/$EDXLINK
           7 - $EDXLINK LINKAGE EDITOR
           8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISORS
           9 - $UPDATE
          10 - $UPDATEH (HOST)
          11 - $PREFIND
          12 - $PASCAL COMPILER/$EDXLINK
          13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

When you select option 7 and press the enter key, the $EDXLINK Parameter Input Menu appears on your screen.

```
$SMM0207: SESSION MANAGER $EDXLINK PARAMETER INPUT MENU--------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN



  EXECUTION PARM ==> *

  ENTER A CONTROL DATA SET NAME,VOLUME OR
  AN ASTERISK (*) FOR INTERACTIVE MODE.

  OUTPUT DEVICE (DEFAULTS TO $SYSPRTR) ==>
```

You can run $EDXLINK in interactive mode. If you choose interactive mode, the system prompts you for information about the object module you want to link-edit. To choose interactive mode, enter an asterisk (*) on the EXECUTION PARM line.

$EDXLINK then displays the following screen:

```
LOADING $JOBUTIL   4P,18:27:16, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 18:28:42 03/15/83 ***

JOB     $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK     89P,18:28:49, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
 DEFAULT VOLUME = EDX002

STMT (?):
```

$EDXLINK prompts you for a control statement. Control statements are the instructions $EDXLINK uses to convert the object modules into load modules.

When using interactive mode, you enter the control statements one at a time. (As you will see later in this chapter, you can write the control statements to a link control data set for execution in noninteractive mode.)

To link-edit a single object module, use the INCLUDE and LINK statements. (You will learn about some of the other control statements later in this chapter.)

The INCLUDE statement indicates which object module to use. (Remember that the object module is the output from $EDXASM, the compiler.) In this example, the object module is OBJECT. This is the only module name you enter next to the INCLUDE statement.

```
LOADING $JOBUTIL   4P,10:27:16, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 10:27:16 00/00/00 ***

JOB     $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK     89P,10:27:18, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
 DEFAULT VOLUME = EDX002

STMT (?): INCLUDE OBJECT,MYVOL
```

Use the LINK statement to name the data set that is the output of $EDXLINK. When you enter the name of this data set, $EDXLINK allocates it. In the following example, the data set is named ADDPGM. It will reside on volume EDX002. The word REPLACE says to replace the program if it already exists on volume EDX002. END tells $EDXLINK not to expect any more statements.

```
LOADING $JOBUTIL    4P,10:27:16, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 10:27:16 00/00/00 ***

JOB     $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK     89P,10:27:18, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
 DEFAULT VOLUME = EDX002

STMT (?): INCLUDE OBJECT,EDX002

STMT (?): LINK ADDPGM,EDX002 REPLACE END
```

The system produces a data set (ADDPGM) that can now be executed on the system. In this example, we link-edited only one object module (OBJECT). The next section shows how to link-edit more than one object module.

If the system indicates (by returning a -1 completion code) that the link-edit was successful, return to the Primary Option Menu to execute your program.

Link-Editing More Than One Object Module

This section shows how to specify that a load module consists of more than one object module. If you divide a large program into modules, those modules can be compiled separately. If you need to make a change to one of the modules, you need to recompile only that module. When you are ready to run the program, you can link-edit the individual modules.

You might also have a function that is common to many of your programs. By making this function a separate module, you could include it wherever needed in your programs.

This section shows how to use both interactive and noninteractive mode to link-edit the modules. All examples show $EDXLINK being used under control of the session manager.

As you learned earlier in this chapter, $EDXLINK LINKAGE EDITOR is option 7 of the Session Manager Program Preparation Option menu.

```
$SMM02  SESSION MANAGER PROGRAM PREPARATION OPTION MENU--------------------
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

      SELECT OPTION ==> 7

            1 - $EDXASM COMPILER
            2 - $EDXASM/$EDXLINK
            3 - $S1ASM ASSEMBLER
            4 - $COBOL COMPILER
            5 - $FORT FORTRAN COMPILER
            6 - $PLI COMPILER/$EDXLINK
            7 - $EDXLINK LINKAGE EDITOR
            8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISORS
            9 - $UPDATE
           10 - $UPDATEH (HOST)
           11 - $PREFIND
           12 - $PASCAL COMPILER/$EDXLINK
           13 - $EDXASM/$XPSLINK FOR SUPERVISORS
```

When you select option 7, the $EDXLINK Parameter Input Menu appears on your screen.

```
$SMM0207: SESSION MANAGER $EDXLINK PARAMETER INPUT MENU--------------------
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN



  EXECUTION PARM ==> *

  ENTER A CONTROL DATA SET NAME,VOLUME OR
  AN ASTERISK (*) FOR INTERACTIVE MODE.

  OUTPUT DEVICE (DEFAULTS TO $SYSPRTR) ==>
```

Link-Editing More Than One Object Module *(continued)*

Using Interactive Mode

You can choose interactive mode or noninteractive mode.

When you choose interactive mode, $EDXLINK displays the following screen:

```
LOADING $JOBUTIL    4P,07:27:16, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 07:27:16 00/00/00 ***

JOB     $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK     89P,07:27:18, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
 DEFAULT VOLUME = EDX002

STMT (?):
```

Including Individual Object Modules

With the INCLUDE statement, you indicate which object modules to use. If the modules reside on the same volume, you can list them on one INCLUDE statement. In the example shown below, the first INCLUDE statment includes four object modules from volume EDX003. The second INCLUDE statment includes two object modules from volume MYVOL.

```
LOADING $JOBUTIL   4P,07:27:16, LP= 9400, PART= 1
REMARK
$EDXLINK *
*** JOB - $EDXLINK - STARTED AT 07:27:16 00/00/00 ***

JOB     $EDXLINK ($SMP0207) USERID=ABCD
LOADING $EDXLINK     89P,07:27:18, LP= 9800, PART= 1

$EDXLINK - EDX LINKAGE EDITOR


$EDXLINK INTERACTIVE MODE
 DEFAULT VOLUME = EDX002

STMT (?): INCLUDE OBJ12,OBJ13,OBJ14,OBJ15,EDX003

STMT (?): INCLUDE SQRT,STDEV,MYVOL
```

After you enter the first INCLUDE statement, $EDXLINK prompts you for another statement. Enter the second INCLUDE statment.

Link-Editing More Than One Object Module *(continued)*

The LINK statement tells the linkage editor what to call the load module and where to put it. In this example, the output object data set will be named PGM1. It will reside on volume EDX003. The word REPLACE says to replace the program if it already exists on volume EDX003. END tells $EDXLINK not to expect any more statements.

```
STMT (?): INCLUDE OBJ12,OBJ13,OBJ14,OBJ15,EDX003

STMT (?): INCLUDE SQRT,STDEV,MYVOL

STMT (?): LINK PGM1,EDX003 REPLACE END

$EDXLINK EXECUTION STARTED
PGM1    ,EDX003 STORED
PROGRAM DATA SET SIZE =      7 RECORDS
COMPLETION CODE =  -1

$EDXLINK ENDED AT 09:33:35

$JOBUTIL ENDED AT 09:33:55

PRESS ENTER KEY TO RETURN
```

Once you enter these statements, $EDXLINK produces a load module (PGM1) that is ready for execution. PGM1 consists of six object modules: OBJ12, OBJ13, OBJ14, OBJ15, SQRT, and STDEV.

Including Overlay Segments

Your program may include overlay segments. (Overlay segments are described in detail in "Reusing Storage using Overlays" on page PG-147.) You use the OVERLAY statement to identify these segments to $EDXLINK.

For example, suppose you had a program made up of a resident segment and two overlays. Assume the name of the resident segment is TESTROOT and the overlays are named TESTSUB1 and TESTSUB2. Your control statements would look like this:

```
$EDXLINK INTERACTIVE MODE
  DEFAULT VOLUME = EDX002

STMT (?): INCLUDE TESTROOT,EDX003

STMT (?): OVERLAY

STMT (?): INCLUDE TESTSUB1,EDX003

STMT (?): OVERLAY

STMT (?): INCLUDE TESTSUB2,EDX003

STMT (?): LINK TEST,EDX003 REPLACE END

$EDXLINK EXECUTION STARTED
TEST    ,EDX003 STORED
PROGRAM DATA SET SIZE =  26
COMPLETION CODE =  -1

$EDXLINK ENDED AT 04:05:35
```

The first INCLUDE statement identifies the resident (or root) portion of the program. The INCLUDE statement following the first OVERLAY statement identifies the first overlay segment. The INCLUDE statement following the second OVERLAY statement identifies the *second* overlay segment.

The LINK statement identifies the object output data set.

Link-Editing More Than One Object Module *(continued)*

Using the Autocall Feature

You can use the AUTOCALL control statement to invoke the autocall feature. You can include up to three autocall data set names on the AUTOCALL statement. Autocall data sets contain a list of object module names and volumes, along with their entry points. Use the autocall option to include modules not explicity included via the INCLUDE statement.

You need to use autocall data sets if, for example, you are link-editing a program that uses $IMAGE subroutines. Some instructions, such as GETEDIT and PUTEDIT, also require that you link-edit with the autocall option.

The following is an example of an autocall data set.

```
PGM1,EDX003   ENTER
PGM2,EDX40    START
PGM3,MYVOL    CALC
**END
```

PGM1, PGM2, and PGM3 are object modules on EDX003, EDX40, and MYVOL. ENTER, START, and CALC are the entry points for the modules. The module names must begin in column one and end with a **END statement.

Enter the AUTOCALL statement just before the LINK statement. This example specifies two autocall data sets: the system-supplied autocall data set ($AUTO on volume ASMLIB) and data set MYAUTO on volume MYVOL.

If you specify more than one AUTOCALL statement, the linkage editor uses the last one.

Suppose you wanted to add an AUTOCALL statement to the previous example. You would enter it like this:

```
$EDXLINK INTERACTIVE MODE
  DEFAULT VOLUME = EDX002

STMT (?): INCLUDE TESTROOT,EDX003

STMT (?): OVERLAY

STMT (?): INCLUDE TESTSUB1,EDX003

STMT (?): OVERLAY

STMT (?): INCLUDE TESTSUB2,EDX003

STMT (?): AUTOCALL $AUTO,ASMLIB MYAUTO,MYVOL

STMT (?): LINK TEST,EDX003 REPLACE END
```

The system would respond as follows:

```
$EDXLINK EXECUTION STARTED
TEST     ,EDX003 STORED
PROGRAM DATA SET SIZE =    26
COMPLETION CODE =    -1

$EDXLINK ENDED AT 04:05:35
```

The linkage editor also prints, on the system printer, the names of the object modules it included. For example:

```
INCLUDE $IMOPEN ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $IMGEN  ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $GPLIST ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $GEER   ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $GEAC   ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $IMDTYPE,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $$RETURN,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
INCLUDE $UNPACK ,ASMLIB FROM $AUTO   ,ASMLIB VIA AUTOCALL
```

Link-Editing More Than One Object Module *(continued)*

Using Noninteractive Mode

Using noninteractive mode means that you do not have to enter control statements each time you link-edit a program.

When you use noninteractive mode, you must enter the name of a primary control data set on the $EDXLINK Parameter Input Menu. The primary control data set contains the control statements to be used by $EDXLINK.

You can create the primary control data set using $FSEDIT. Then enter control statements into the data set.

The following is an example of a primary control data set. Control statements must begin in column 1. This data set includes comment statements. A comment statement begins with an asterisk (*).

```
* PLOT PROGRAM INCLUDES
*
INCLUDE PLOTXY,MYVOL
INCLUDE PLOTXX,MYVOL
INCLUDE PLOTYY,MYVOL
INCLUDE PLOTYX,MYVOL
*
* PERFORM AUTOCALL PROCESSING USING:
*
AUTOCALL MYAUTO,MYVOL $AUTO,ASMLIB
*
* PERFORM THE LINK
*
LINK PLOT,MYVOL REPLACE END
```

After entering these statements into the data set, you would then specify the name of this data set next to "EXECUTION PARM" on the $EDXLINK Parameter Input Menu. In this example, the data set is LINK1 on volume EDX003.

```
$SMM0207: SESSION MANAGER $EDXLINK PARAMETER INPUT MENU--------------------
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN



  EXECUTION PARM ==> LINK1,EDX003

  ENTER A CONTROL DATA SET NAME,VOLUME OR
  AN ASTERISK (*) FOR INTERACTIVE MODE.

  OUTPUT DEVICE (DEFAULTS TO $SYSPRTR) ==>
```

The primary control data set may also refer to a secondary control data set. The secondary control data set contains additional control statements. These control statements can be common control statements that are used frequently for many different link-edits. You use the COPY control statement to refer to these secondary data sets. For example:

```
INCLUDE  ASMOBJ,EDX003
COPY  CTRL,EDX40
LINK  PGM3,EDX40 REPLACE END
```

The linkage editor includes object module ASMOBJ on volume EDX003, copies additional control statements from data set CTRL on volume EDX40, gives the load module the name PGM3, and puts it on volume EDX40.

For more information on specifying primary and secondary control statement data sets, refer to *Operator Commands and Utilities Reference.*

Prefinding Data Sets and Overlays

You can locate data sets and overlay programs before you load a program by using the $PREFIND utility. You can improve program performance by using $PREFIND.

You should use $PREFIND if:

- The program uses a large number of data sets
- The program loads several overlay programs
- You load the program frequently

For information on how to use the $PREFIND utility, refer to *Operator Commands and Utilities Reference.*

Notes

Chapter 6. Executing a Program

After you have compiled and link-edited a program, you are ready to run (or *execute*) it.

This chapter shows how to execute a program. You can execute a program in any of the following ways:

- You can load the program with the $L operator command.
- You can use the $JOBUTIL utility.
- You can use the session manager.
- You can submit the program from another program.
- You can use the $SUBMIT utility.

This chapter describes how to use the session manager to execute a program and how to submit a program from another program. For information on how to use the $L operator command or the $JOBUTIL utility or the $SUBMIT utility, refer to *Operator Commands and Utilities Reference*.

Executing a Program with the Session Manager

To execute your program, select option 6 (EXEC PROGRAM/UTILITY) on the Primary Option Menu.

```
$SMMPRIM: SESSION MANAGER PRIMARY OPTION MENU ----------------------------
ENTER/SELECT PARAMETERS:                                 PRESS PF3 TO EXIT

                                                             11:42:07
     SELECT OPTION ==> 6                                     10/24/82
                                                             ABCD

           1 - TEXT EDITING
           2 - PROGRAM PREPARATION
           3 - DATA MANAGEMENT
           4 - TERMINAL UTILITIES
           5 - GRAPHICS UTILITIES
           6 - EXEC PROGRAM/UTILITY
           7 - EXEC $JOBUTIL PROC
           8 - COMMUNICATION UTILITIES
           9 - DIAGNOSTIC AIDS
```

The Execute Program/Utility menu appears. Enter the program name (ADDPGM) and volume (EDX002) next to PROGRAM/UTILITY (NAME,VOLUME). Then type an asterisk in the DATA SET 1, DATA SET 2, and DATA SET 3 fields and press the enter key.

```
$SMM06  SESSION MANAGER EXECUTE PROGRAM/UTILITY----------------------------
ENTER/SELECT PARAMETERS:                               PRESS PF3 TO RETURN

 PROGRAM/UTILITY (NAME,VOLUME) ==> ADDPGM,EDX002

 PARAMETERS ===>

 DATA SET 1 (NAME,VOLUME / * = DS1 NOT USED) ===> *
 DATA SET 2 (NAME,VOLUME / * = DS2 NOT USED) ===> *
 DATA SET 3 (NAME,VOLUME / * = DS3 NOT USED) ===> *

 NOTE: IF A DATA SET (DS1, DS2 OR DS3) IS NOT USED,
       AN ASTERISK (*) MUST BE ENTERED IN THE DATA SET FIELD.
```

Putting asterisks in the DATA SET fields means either of two things. Either the program does not use any data sets or the program specifies the data sets with the DS operand. For example, the PROGRAM for program ADDPGM might look like this:

```
BEGIN   PROGRAM   ST
```

or this:

```
BEGIN   PROGRAM   ST,DS=((MASTER,EDX003),(UPDATES,MYVOL),(NEWMAST,EDX40))
```

The following screen then appears on the terminal:

```
LOADING $JOBUTIL    4P,11:48:21, LP= 9400, PART= 1
REMARK
EXECUTE  PROGRAM/UTILITY: ADDPGM
*** JOB - ADDPGM - STARTED AT 11:48:22 00/00/00 ***

JOB      ADDPGM ($SMP06) USERID=ABCD
LOADING ADDPGM        2P,11:48:23, LP= 9800, PART= 1
ENTER NUMBER:
```

Specifying Data Sets

You can specify data sets in one of six ways:

1. In the DS= operand of a PROGRAM instruction.

2. In the DS= operand of a LOAD instruction.

3. With the $L operator command.

4. During execution of some system utility programs.

5. On the Execute Program/Utility menu.

6. With the DS command of the $JOBUTIL utility.

You identify a data set by specifying:

1. The data set name (dsname)

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

The format for a data set specification is:

```
dsname,volume
```

Volume is optional. If you omit volume, the system assumes that the data set resides on the volume from which you performed an IPL. Definitions of dsname and volume are:

dsname An alphameric character string of eight characters. When you specify fewer than eight characters, the system adds blanks to the right to complete the string.

volume An alphameric character string of six characters. To locate the volume, the appropriate TAPE or DISK statement must be in the system I/O definition. You must initialize the disk or diskette with the $INITDSK utility and tapes with the

Executing a Program with the Session Manager *(continued)*

$TAPEUT1 utility. When you specify fewer than six characters, the system adds blanks to the right to complete the string.

To specify up to three data sets on the Execute Program/Utility menu, enter the data set name and volume as in the following example:

```
$SMM06  SESSION MANAGER EXECUTE PROGRAM/UTILITY----------------------------
ENTER/SELECT PARAMETERS:                                  PRESS PF3 TO RETURN

 PROGRAM/UTILITY (NAME,VOLUME) ==> ADDPGM,EDX002

 PARAMETERS ===>

 DATA SET 1 (NAME,VOLUME / * = DS1 NOT USED) ===> MASTER,EDX003
 DATA SET 2 (NAME,VOLUME / * = DS2 NOT USED) ===> UPDATES,MYVOL
 DATA SET 3 (NAME,VOLUME / * = DS3 NOT USED) ===> NEWMAST,EDX40

 NOTE: IF A DATA SET (DS1, DS2 OR DS3) IS NOT USED,
       AN ASTERISK (*) MUST BE ENTERED IN THE DATA SET
       FIELD.
```

The PROGRAM statement for program ADDPGM might look like this:

```
BEGIN   PROGRAM   ST,DS=(??,??,??)
```

If a program requires less than three data sets, type an asterisk (*) next to the data set(s) not used.

Submitting a Program from Another Program

A program can submit one or more programs to the EDX job processor. The *job queue processor* executes the programs independently of the program that submitted them.

The following example shows how one program can submit programs CALC on volume EDX003 and UPDATE on volume MYVOL.

```
     BEGIN    PROGRAM  START
     START    EQU      *
               .
               .
1             LOAD     $SUBMITP,SUBPARM1,LOGMSG=NO,EVENT=SUBEND
2             WAIT     SUBEND
3             IF       (SUBEND,NE,-1)
                 PRINTEXT  'ERROR LOADING CALC',SKIP=1
               ENDIF
               .
               .
4             LOAD     $SUBMITP,SUBPARM2,LOGMSG=NO,EVENT=SUBEND
               WAIT     SUBEND
               IF       (SUBEND,NE,-1)
                 PRINTEXT  'ERROR LOADING UPDATE',SKIP=1
               ENDIF
               .
               .
               PROGSTOP
     SUBEND    ECB
     SUBPARM1 EQU      *
5             DATA     C'SJ'
6             DATA     X'0002'
7             DATA     CL8'JOB01'
8             DATA     CL6'EDX40'
9             DATA     A(JOBNO)
     SUBPARM2 EQU      *
               DATA     C'SJ'
               DATA     X'0002'
               DATA     CL8'JOB02'
               DATA     CL6'EDX40'
               DATA     A(JOBNO)
10 JOBNO      DATA     F'0'
               ENDPROG
               END
```

1 Submit a job to the job queue. Point to a parameter list called SUBPARM1, and identify the event to be posted when the job has been submitted (EVENT=SUBEND).

2 Wait for the job to be submitted to the job queue.

3 Test for successful completion (-1) of the submit.

Submitting a Program from Another Program *(continued)*

4 Submit a job to the job queue. Point to a parameter list called SUBPARM2, and identify the event to be posted when the job has been submitted (EVENT=SUBEND).

5 Specify that the job is to be submitted (SJ).

6 Specify the priority of the job (0002).

7 Identify the name of the data set that contains the job stream processor commands (JOB01).

8 Specify the volume that contains JOB01 (EDX40).

9 Specify the address of the field in which the system will put the job number (JOBNO).

10 Reserve storage for the system to put the job number.

The data set called JOB01 contains job stream processor commands. It might look like the following:

```
JOB  JOB01
PROGRAM  CALC,EDX003
EXEC
EOJ
```

The PROGRAM command refers to a program called CALC on volume EDX003.

The data set called JOB02 contains job stream processor commands. It might look like the following:

```
JOB  JOB02
PROGRAM  UPDATE,MYVOL
EXEC
EOJ
```

The PROGRAM command refers to a program called UPDATE on volume MYVOL.

Chapter 7. Finding and Fixing Errors

Up to this point, you have written, compiled, and link-edited your program. However, the program may not run as you expect it to. Steps may be out of sequence or the program may come up with the wrong answers. In other words, you have problems with your program's logic.

The program also may not run to a successful conclusion. An exception condition may occur that interrupts the execution of a program.

The $DEBUG utility assists you in determining logic errors. The task error exit routine is one of the tools you can use to diagnose exception conditions.

Determining Logic Errors in a Program

This section tells you how to locate and fix logic errors in your program by using the $DEBUG utility. $DEBUG can work from terminals; you do not have to use the console. $DEBUG has commands that allow you to:

- Stop execution at one or more specific places in a program. The places where you choose to stop a program are called breakpoints.

- Set up a trace routine. A trace routine allows you to step through program instructions one at a time. You must specify one or more parts of the program you wish to trace (called a trace range). Each time the program executes an instruction within any of the specified trace ranges, the terminal displays a message identifying the task name and the instruction address just executed. You can stop program execution after each instruction executes within a trace range.

- List additional registers and storage location contents while the program is stopped at a breakpoint or at an instruction within a trace range.
- Change the contents of storage locations, registers, data, or instructions.
- Restart program execution. You can restart execution at the breakpoint or trace range address where it is currently stopped or you may specify another instruction address.

Creating and Running the Program

This section shows an EDL program that has a logic error in it. It shows briefly how to enter, compile, link-edit, and run (execute) the program.

Perform the following steps using the session manager. Give the program the name ADD10.

1. Enter the following program on your terminal exactly as shown.

```
ADD10     PROGRAM    STPGM
STPGM     GETVALUE   COUNT,'ENTER NUMBER: '
LOOP      DO         10,TIMES
            ADD      COUNT,SUM
          ENDDO
          PRINTEXT   'RESULT='
          PRINTNUM   SUM
          PROGSTOP
 COUNT    DATA       F'0'
 SUM      DATA       F'0'
          ENDPROG
          END
```

   This program is supposed to take a number entered on a terminal and add it to itself 10 times. For example, if you enter the number 10, you should get the response: RESULT=100. However, because of a program logic error, you will not get the expected answer when you run the program.

2. Now compile the program. If you have any problems, see Chapter 4, "Compiling a Program." Save the compiler listing. You will need it when you run $DEBUG.

3. Next, link-edit your program. If you have any problems, see Chapter 5, "Preparing an Object Module for Execution."

4. Run the program. If you have any problems, see Chapter 6, "Executing a Program."

When the prompt ENTER NUMBER appears, enter the number 10.

```
ENTER NUMBER: 10
RESULT=      0
```

Because this program has a logic error, the answer returned is 0. The expected result was 100.

Debugging and Fixing the Program

This section describes how to use $DEBUG to find and correct a logic error.

Loading $DEBUG

To start debugging the program, do the following:

1. End the session manager. You cannot run $DEBUG while the session manager is active. One way to load $DEBUG is with the $L operator command.

2. Enter the following:

```
> $L $DEBUG
```

The following message appears, telling you that $DEBUG is being loaded.

```
LOADING $DEBUG      31P,00:48:05,  LP=  9E00,  PART= 1
```

3. Then $DEBUG asks for the name of the program to be debugged. Respond as follows:

```
PROGRAM NAME: ADD10,EDX002
```

$DEBUG displays the following information:

```
LOADING ADD10        2P,00:48:12,  LP=  BD00,  PART= 1

REQUEST "HELP" TO GET LIST OF DEBUG COMMANDS
ADD10    STOPPED AT   0034
```

These messages tell you:

- the load point (LP=) of the program.
- the partition where $DEBUG loaded the program.
- that $DEBUG set a breakpoint and stopped the program at address 0034, which is the first executable instruction.

Note that you can also enter HELP to see a list of the available $DEBUG commands.

Determining Logic Errors in a Program *(continued)*

$DEBUG Commands

Both $DEBUG and the program have been loaded into partition 1. The program has stopped and $DEBUG is waiting for a command. To see a list of the $DEBUG commands:

1. Press the attention key.

2. Enter HELP.

The list of $DEBUG commands appears on the screen.

```
> HELP
THE FOLLOWING COMMANDS ARE AVAILABLE:

HELP     - LIST DEBUG COMMANDS
WHERE    - DISPLAY TASK STATUS
LIST     - DISPLAY STORAGE OR REGISTERS
PATCH    - MODIFY STORAGE OR REGISTERS
QUALIFY  - MODIFY BASE ADDR
AT       - ESTABLISH BREAKPOINTS
OFF      - REMOVE BREAKPOINTS
GO       - START TASK PROCESSING
POST     - POST EVENT OR PROCESS INTERRUPT
PRINT    - DIRECT LISTING TO PRINTER
BP       - LIST BREAK POINTS
GOTO     - CHANGE EXECUTION SEQUENCE
END      - TERMINATE DEBUG FACILITY
```

Use the $DEBUG commands to:

- Set breakpoints and trace ranges (AT).
- List breakpoints and trace ranges (BP).
- End $DEBUG (END).
- Restart a stopped task (GO).
- Restart a stopped task at a different instruction (GOTO).
- List $DEBUG commands (HELP).
- Display storage or register contents (LIST).
- Remove breakpoints and trace ranges (OFF).
- Change storage or register contents (PATCH).
- Start a task waiting for an event or process interrupt (POST).
- Direct output to another terminal (PRINT).
- Change the base address (QUALIFY).
- Display the current status of each task (WHERE).

You can enter any of the commands by pressing the attention key and entering the command name. $DEBUG then prompts for the command parameters. For example, if you want to set a breakpoint, enter the AT command. $DEBUG then prompts for the parameters as shown below.

```
> AT
OPTION(*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDRESS: 4C
LIST/NOLIST: LIST
OPTION(*/ADDR/R0...R7/#1/#2/IAR/TCODE):
LENGTH: 1
MODE(X/F/D/A/C): X
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

This command sets a breakpoint at address 4C. It requests that $DEBUG print the contents of register 1 (one word) in hexadecimal. STOP tells $DEBUG to stop at address 4C.

For detailed syntax descriptions, refer to each individual command in the *Operator Commands and Utilities Reference.*

You can also enter a command and its parameters without going through the prompts. For example:

```
> AT ADDR 4C L #1 1 X S
```

gives you the same results.

Determining Logic Errors in a Program *(continued)*

Finding the Errors

Now that you have loaded $DEBUG, specified your program name, and reviewed the $DEBUG commands, you are ready to start finding the logic errors in your program. You should have a listing of the program before you start. Then follow these steps:

1. Use the AT command to set a breakpoint to stop the program after the GETVALUE executes (address 004C). Respond to the prompts as follows:

```
> AT
OPTION(*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: 004C
LIST/NOLIST: NOLIST
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

   The breakpoint to stop after the GETVALUE instruction executes is now set.

2. Enter a GO command and, when prompted, enter the number 10.

```
> GO
      1 BREAKPOINT(S) ACTIVATED
ENTER NUMBER: 10
ADD10     STOPPED AT   004C
```

   Program execution has stopped at the instruction labeled LOOP. The GETVALUE instruction has executed.

   To check to see if the program read the data correctly, use the LIST command to display data field COUNT at address 0074.

3. Enter a LIST command and respond as follows:

```
> LIST
OPTION(*/ADDR/RO...R7/#1/#2/IAR/TCODE): ADDR
ADDRESS: 0074
LENGTH: 1
MODE(X/F/D/A/C): D
  0074 D' 0010'
```

   The LIST command shows that 0074 contains 10, the correct input. This indicates proper logic to this point.

   The next set of instructions is the DO loop. Set another breakpoint to stop the program after execution of the DO loop at address 005E.

4. Enter an AT command and respond as follows:

```
> AT
OPTION(*/ADDR/TASK/ALL): ADDR
BREAKPOINT ADDR: 005E
LIST/NOLIST: NOLIST
STOP/NOSTOP: STOP
      1 BREAKPOINT(S) SET
```

The breakpoint to stop after the DO loop instructions executes is now set.

5. Enter a GO command and the following occurs:

```
> GO
      1 BREAKPOINT(S) ACTIVATED
ADD10     STOPPED AT   005E
```

At this point, the data field SUM at address 0076 should contain the number 100.

To check to see if the data field SUM contains the proper number, use the LIST command to display data field SUM at address 0076.

6. Enter a LIST command and respond as follows:

```
> LIST
OPTION(*/ADDR/R0...R7/#1/#2/IAR/TCODE): ADDR
ADDRESS: 0076
LENGTH: 1
MODE(X/F/D/A/C): D
  0076 D' 0000'
```

The LIST command shows that this field contains zero. This means that the DO loop or the ADD instruction in the DO loop is incorrect. If you examine these instructions, you will see that the DO loop is correct. However, The ADD instruction has a logic error. In order to receive the proper answer, the COUNT field should be added to the SUM field. The operands are backwards. The DO loop executes the ADD instruction 10 times but is adding SUM to COUNT, causing the SUM field to remain 0.

Fixing the Problem

To verify that this is the problem without having to recompile and link-edit the program, you can use the PATCH command of $DEBUG for a temporary fix. This fix is good only for one execution of the program. PATCH only fixes the copy of the program loaded by $DEBUG. It does not fix the program on your volume. Once you have verified that the fix is correct, you can then change the program on your volume.

To verify that the problem is the ADD instruction, do the following:

1. Find address 0052 on your compiler listing. This line contains the ADD instruction. The entire line looks like this:

   ```
   0052    0032 0074 0076                           ADD          COUNT,SUM
   ```

   The address of the instruction is 0052. The operation code (0032) does not change. The next two words, 0074 and 0076, are the addresses of data fields COUNT and SUM.

   To fix the logic error, change the instruction to look as follows:

   ```
   0052     0032 0076 0074
   ```

2. Enter a PATCH command and respond to the prompts as follows:

```
> PATCH
OPTION( */ADDR/RO...R7/#1/#2/IAR/TCODE): ADDR
ADDRESS: 0054
LENGTH: 2
MODE(X/F/D/A/C): A
NOW IS
 0054 A'  0074 0076'
DATA:  0076 0074
NEW DATA
 0054 A'  0076 0074'
YES/NO/CONTINUE: YES
```

   The program is now patched. When it executes, it will add COUNT to SUM to arrive at the expected result. You can test the change by reexecuting the program.

   To reexecute the program, you have to know two things: the address where the program is currently stopped (005E) and the address of the first executable instruction (0034). Then you can use the GOTO command to restart the program at the first executable instruction.

3. Enter a GOTO command as shown:

```
> GOTO 005E 0034
 1 BREAKPOINT(S) ACTIVATED
ADD10    STOPPED AT 0034
```

4. The program is now at the beginning. To test it, set all the breakpoints off so that the program will run to completion.

Enter the following:

```
> OFF ALL
```

5. Now enter a GO command and respond to the prompts as follows:

```
> GO
ENTER NUMBER: 10
RESULTS=      100
ADD10   ENDED AT 00:27:56
```

This time you received the expected result of 100. You have verified that the logic error was the ADD instruction.

Ending $DEBUG

Now that you have found and fixed the logic error in your program, use the END command to terminate $DEBUG. Enter the following:

```
> END
```

When $DEBUG ends, your program remains in storage with all of its tasks active and operating if it has not already ended. In our example, however, the program has ended.

To make the fix permanent, change your source program (see Chapter 3, "Entering a Source Program" on page PG-59), recompile it (seeChapter 4, "Compiling a Program" on page PG-69), and link-edit your object code module (see Chapter 5, "Preparing an Object Module for Execution" on page PG-81).

Using Return Codes to Diagnose Problems

This section describes how to use the return codes to diagnose problems.

Many EDL instructions return a code to indicate whether or not they execute successfully. Each time EDX executes one of these instructions, it stores a code, called a *return code*, in the first two words, called *task code words*, of the task control block (TCB). You can access the TCB by referencing the task name.

In the following example, the instructions at label ERRTEST compare the return code of the READTEXT instruction with the successful return code (-1).

```
BEGIN    PROGRAM  START
         .
         .
         .
         READTEXT  NAME,'ENTER NAME: ',SKIP=4,MODE=LINE
ERRTEST  MOVE      TASKRC,BEGIN
         IF        (TASKRC,NE,-1),GOTO,CHECK
         ENDIF
         .
         .
         .
CHECK    PRINTEXT  'ERROR IN READING NAME',SKIP=1
         PRINTNUM  TASKRC
         GOTO      END
         .
         .
         .
END      PROGSTOP
TASKRC   DATA      F'0'
         ENDPROG
         END
```

You must test the return code before executing any other instruction because the system may overlay the task code word with the return code of the next instruction.

Task Error Exit Routines

This section describes the facilities provided by the system in the event that an exception occurs. These are the supervisor facility and the system-supplied task error exit routine.

When an exception occurs, the supervisor takes certain actions. What action it takes depends on whether or not you have coded a task error exit routine in your program. If your program does not have a task error exit routine, the supervisor simply writes a program check message on $SYSLOG, and terminates the program. If your program has a task error exit routine, either the one supplied by the system or your own, the supervisor does the following:

1. Stores the hardware status at the time of the exception in a block of storage designated by the task.

2. Passes control to the task at its task error exit entry point.

At this point, the task error exit routine gains control. The next section discusses only the system-supplied routine. However, remember that, if necessary, you can substitute your own routine. (For information on writing your own task error exit routine, refer to *Customization Guide*.)

Notes:

1. A task error exit routine is a part of the task it serves. The supervisor passes control to it at the task level; it is not a subroutine of the supervisor's error handler.

2. The registers (including the EDL software registers, #1 and #2) used by the error exit routine are those normally used by the task.

3. To resume executing the task following corrective action by task error exit, branch (if in Series/1 instruction mode) or GOTO (if in EDL mode) the appropriate location.

4. If the error exit is unable to recover from the exception, it should issue a PROGSTOP instruction.

The System-Supplied Task Error Exit Routine ($$EDXIT)

A task error exit routine named $$EDXIT is available on volume ASMLIB. This routine:

- Captures relevant data from the program header, task control block, and hardware status area when an exception occurs

- Formats and prints this data on $SYSLOG and $SYSPRTR

- Displays an error message on the loading terminal

Using $$EDXIT

To use the supplied routine, you must:

- Code $$EDXIT as the value of the ERRXIT keyword parameter of each PROGRAM and TASK statement in your program. For example:

```
AB PROGRAM ....,ERRXIT=$$EDXIT
   .
   .
CD TASK    ....,ERRXIT=$$EDXIT
```

- Declare the label $$EDXIT to be an EXTRN.

```
   EXTRN  $$EDXIT
```

The task error exit routine is included in the autocall list $AUTO on volume ASMLIB. It is automatically included when you link-edit any program that references $$EDXIT. A separate INCLUDE statement is not required for $$EDXIT in the LNKCTRL data set. All you need to do is code $AUTO,ASMLIB as the autocall data set on the AUTOCALL statement of $EDXLINK.

The following example shows what $$EDXIT prints on $SYSLOG and $SYSPRTR. It shows that a program check has occurred in an application program named PCHECK. The numbers to the left of both columns correspond to the explanations that follows.

```
        ****************************************
        * WARNING!! AN EXCEPTION HAS OCCURRED!! *
        ****************************************

1 PROGRAM NAME             = PCHECK      9 PSW =  8002
2 PROGRAM VOLUME           = EDXWRK          IAR =  3124
3 PROGRAM LOAD POINT       = 0000            AKR =  0440
4 ADDRESS OF ACTIVE TCB    = 016C            LSR =  00D0
5 ADDRESS OF CCB           = 1802            R0 (WORK REGISTER)  = 0096
    NUMBER OF DATA SETS      =    1       10 R1 (EDL INSTR ADDR) = 00E7
    NUMBER OF OVERLAYS       =    0       11 R2 (EDL TCB ADDR)   = 016C
6 $TCBADS                  =  0004           R3 (EDL OP1 ADDR)   = 00E7
    ADDRESS OF FAILURE                          R4 (EDL OP2 ADDR)   = 00B2
7  (REL. TO PGM LOAD POINT =  00E7       12 R5 (EDL COMMAND)     = 0000
    DUMP OF FAIL ADDRESS                        R6 (WORK REGISTER)  = 0000
8  00E6: 0000 0028 0028 3635                 R7 (WORK REGISTER)  = 0000
    $TCBCO =        -1 DEC;  FFFF HEX             #1 =  0000
    $TCBO2 =         0 DEC;  0000 HEX             #2 =  0000

    PSW ANALYSIS:

    SPECIFICATION CHECK
    TRANSLATOR ENABLED
```

Explanation:

1 Name of the active program

2 Name of the volume where the program resides

3 The load point of the program

4 Address of the active TCB when the exception occurred

5 Address of the CCB (terminal that loaded the program)

6 Address key where program is loaded if not doing cross-partition move or the target address key if doing a cross-partition move

7 Address of the instruction that caused the program check

8 Dump of the instruction that caused the program check

9 Indicates the type of exception that occurred

10 Usually points to the EDL instruction address

11 Usually contains the EDL TCB address

12 Usually contains the op code of the EDL instruction that was being executed

The following message appears on the loading terminal when the program check occurs:

```
A MALFUNCTION HAS OCCURRED -- CALL SYSTEM PROGRAMMER
```

Notes:

1. If you are executing either a combination of EDL instructions and Series/1 instructions or all Series/1 instructions, the registers may not contain this information.

2. You can restart the program by writing your own error exit routine to reload it.

$$EDXIT provides you with information about the program, task, and hardware status when an exception occurs. You can extend the capabilities of $$EDXIT so that it will also evaluate the information and make an appropriate response. For more information on writing your own task error exit routine, refer to *Customization Guide*.

Notes

Chapter 8. Reading and Writing Data from Screens

The Event Driven Executive allows you to read and write data from a screen that appears on a terminal. A person at a terminal can supply data to a program and the program can display information on the terminal screen.

EDX allows you to use two types of screens: roll screens and static screens.

This chapter describes:

- When to use roll screens
- When to use static screens
- Reading and writing rolls screens
- Reading and writing static screens

The chapter shows how to write a program to read five data items from a screen and write them back to the screen. The chapter shows how to use each kind of screen (roll and static).

You can generally code terminal programs using either roll or static screens. However, each screen offers distinct advantages for certain types of programs.

When to Use Roll Screens

A *roll screen* is similar to a typewriter. The system reads or writes data line-by-line, starting with line 0 at the top of the screen and ending with line 23 at the bottom of the screen. You can use roll screens to read or write a single data item.

A program that uses roll screens usually prompts the operator for data, waits for an operator response, and checks the validity of the input data. Roll screens are best suited for application programs in which:

- A simple question-and-answer dialogue occurs between program and operator.
- A single line is sufficient for each response.
- No Program Function (PF) keys are required.
- An incorrect response requires only a reprompt.
- You want to use a minimum of processor storage.

In addition, the terminal may support roll screens only.

Roll screen dialogue is relatively easy to code and requires little program preparation. You can code prompts in a tree structure where the choice of the next prompt depends on the reply to past prompts.

You can print more than one line of text to introduce a prompt. For example, you might want to offer the choice of several programs to be loaded, each of which may choose to continue the dialogue at the same terminal. You can also display more than one line of text in a program reply.

When to Use Static Screens

A *static screen* is similar to a page of information. The system reads or writes an entire screen at once. You can use static screens to read or write several data items at one time.

Programming for static screens involves managing the entire screen as a series of protected and unprotected fields.

A *protected field* is an area that contains an operator prompt or an input field name. It is protected from being accidentally changed by the operator.

An *unprotected field* is an area that is to be filled in by the operator.

When to Use Static Screens *(continued)*

Static screens are best suited for programs in which:

- The dialogue involves a series of full screens.
- More than one line of response may be required.
- You need to manipulate the cursor.
- The program must recognize Program Function (PF) keys.
- You need attribute characters such as blinking and non-display.
- The unprotected fields may be scattered across the screen and interspersed with the protected fields.
- Many related data fields are to be entered at one time.
- Medium to large amounts of data accompany each prompt, operator response, or program reply.

You can manage static screens most easily by using the $IMAGE utility to define your screens. $IMAGE places the screens on direct access storage. The program then can read them into processor storage. $IMAGE subroutines and terminal I/O statements allow you to read the screen into the application program, display it at the terminal, position the cursor, scatter read or write unprotected fields, and wait for a response.

Reading and Writing One Line at a Time

Reading and writing a single line from a terminal screen involves reading the data item from the screen and writing or *displaying* the data item on the screen.

To read and write to a roll screen:

1. Reserve storage for data.
2. Read a data item.
3. Write a data item.

Reading and Writing One Line at a Time *(continued)*

Reserving Storage for the Data

To reserve storage for a data item that you will read, you must know its maximum length. To reserve storage for a text string of 30 characters, use the TEXT statement as follows:

```
NAME     TEXT      LENGTH=30
```

The name of the storage is NAME. The next section describes how to put a data item into NAME.

Reading a Data Item

To read a data item from a roll screen, you can use either the READTEXT or GETVALUE instruction. The READTEXT instruction allows you to read a text string. The GETVALUE instruction allows you to read one or more numbers.

To read a data item into a storage area, use the READTEXT instruction as follows:

```
READTEXT NAME,'NAME:',SKIP=1,MODE=LINE
```

The instruction displays the prompt **NAME**: and the system waits for a response. When the operator enters a name and presses the enter key, the system stores the text string in an area called NAME.

The operand SKIP=1 causes the system to skip one line before displaying the prompt. The operand MODE=LINE allows blanks in the response. Since most names contain at least one blank, you must code MODE=LINE to read the entire name.

Writing (Displaying) a Data Item

Writing (or *displaying*) a data item involves transferring the data item from storage to the terminal screen. You can use either the PRINTNUM or PRINTEXT instruction to transfer data to the terminal screen. The PRINTNUM instruction transfers one or more numbers. The PRINTEXT instruction transfers a text string.

To display the data item called NAME, use the PRINTEXT instruction as follows:

```
PRINTEXT NAME,SKIP=3
```

The operand SKIP=3 causes the system to skip three lines before displaying NAME.

Example

Prompt the operator for five data items: name, address, city, state, and zip code. Then display the five data items. Read from and write to the terminal that loaded the program.

```
1  TEST      PROGRAM   BEG
   BEG       EQU       *
2            READTEXT  NAME,'          NAME:',SKIP=1,MODE=LINE
3            READTEXT  ADDR,'       ADDRESS:',MODE=LINE
             READTEXT  CITY,'          CITY:',MODE=LINE
             READTEXT  ST,'         STATE:'
             READTEXT  ZIP,'          ZIP:'
4            PRINTEXT  NAME,SKIP=3
5            PRINTEXT  ADDR,SKIP=1
             PRINTEXT  CITY,SKIP=1
6            PRINTEXT  ST,SPACES=1
             PRINTEXT  ZIP,SPACES=2
             PROGSTOP
   NAME      TEXT      LENGTH=30
   ADDR      TEXT      LENGTH=30
   CITY      TEXT      LENGTH=30
   ST        TEXT      LENGTH=2
   ZIP       TEXT      LENGTH=5
             ENDPROG
             END
```

1 Begin the program and execute the instruction at label BEG.

2 Prompt the operator for name and read the operator's response. Allow spaces in the name (MODE=LINE), skip one line (SKIP=1), and store the response in NAME.

3 Prompt the operator for address and read the operator's response. Allow spaces in the name (MODE=LINE) and store the response in ADDRESS. Because the program writes to a roll screen, the prompt appears one line below the prompt for name.

4 Display the data item in NAME. Skip three lines before displaying (SKIP=3).

5 Display the data item in ADDR. Skip to the beginning of the next line before displaying (SKIP=1).

6 Display the data item in ST. Leave one blank space to the right before displaying (SPACES=1).

Reading and Writing One Line at a Time *(continued)*

Executing the Example

If you entered, compiled, link-edited, and loaded the example, the system would issue a prompt for each data item. After entering each data item, press the enter key. After you enter the last data item (zip code) and press enter, the system displays the data items.

After you enter all five data items, the screen might look like this:

```
        NAME:ROSE PETERSON
     ADDRESS:11 CYPRESS CREEK RD.
        CITY:SALINA
       STATE:KA
         ZIP:45367
```

When you press the enter key, the program displays the name and address as follows:

```
ROSE PETERSON
11 CYPRESS CREEK RD.
SALINA KA  45367
```

Note: Even though CITY is 30 characters long, the system displays only the actual length of the data.

Two Ways to Use Static Screens

Reading and writing an entire screen at once involves using *static screens*. The Event Driven Executive provides two methods to define static screens.

The first method requires that the format of the screen be defined within the program. Any change to the screen requires a change to the program.

In addition, programs that use this method are usually *not* device independent. In other words, a program that contains instructions that define a static screen may execute successfully on a 4978 or 4979 terminal and *not* execute on a 3101 terminal.

The section called "Coding the Screen within a Program" on page PG-122 describes the first method.

The second method for defining screens involves defining the screen with the $IMAGE utility and saving it in a data set. This method allows more than one program to use the same screen. In addition, a change to the screen does not necessarily require a change to each program that uses it.

Finally, you can write programs that are device independent. You can write programs that execute successfully on 4978, 4979, or 3101 terminals.

The section called "Writing the Screen Image to a Data Set" on page PG-127 describes the second method.

For more information on coding static screens, see Appendix C, "Static Screens and Device Considerations" on page PG-269.

Coding the Screen within a Program

This section describes reading data from and writing data to a static screen. Instructions in the program create the static screen.

For more information on static screens, refer to Appendix C, "Static Screens and Device Considerations" on page PG-269.

To read and write to a screen that you define in a program, do the following:

1. Define the screen as static.
2. Get exclusive access to the terminal.
3. Erase the screen.
4. Reserve storage for data.
5. Prompt the operator for a data item.
6. Position the cursor.
7. Wait for a response.
8. Read a data item.
9. Write a data item.

Defining a Screen as Static

To define a screen as a static screen, use the IOCB statement as follows:

```
TERM   IOCB   SCREEN=STATIC
```

This statement defines the loading terminal as a static screen. The label TERM defines the name you will use in other instructions in the program.

For information on defining logical screens, see Appendix C, "Static Screens and Device Considerations" on page PG-269.

Getting Exclusive Access to the Terminal

Before you can use a terminal as a static screen, you must get exclusive access to it. Use the ENQT instruction as follows:

```
ENQT   TERM
```

The operand TERM is the name you used to define the terminal in an IOCB instruction.

Erasing the Screen

Before you code instructions that prompt the operator for data, you should erase the screen. To erase the screen, use the ERASE instruction as follows:

```
ERASE     MODE=SCREEN,TYPE=ALL,LINE=0
```

The operand LINE=0 tells the system to begin erasing on line 0 (the first line) of the screen. The operand MODE=SCREEN causes the system to erase to the end of the screen. The operand TYPE=ALL allows the system to erase both protected and unprotected data.

Reserving Storage

To reserve storage for a data item that you read, you must know its maximum length. To reserve storage for a text string of 30 characters, use the TEXT statement as follows:

```
NAME     TEXT      LENGTH=30
```

The name of the storage is NAME. The READTEXT instruction transfers the data item containing the name into this area of storage.

Prompting the Operator for a Data Item

One way you can display information on a static screen is by issuing PRINTEXT instructions. For example, to prompt the operator for a name, use the PRINTEXT instruction as follows:

```
PRINTEXT 'NAME: ',LINE=1,PROTECT=YES
```

The instruction displays the prompt **NAME:**. The operand LINE=1 causes the system to display the prompt on the second line of the screen. (The lines on a screen are numbered 0-23 and the columns are numbered 0-79.) The operand PROTECT=YES causes the prompt **NAME**: to be protected. A *protected* is a field that cannot be changed by the operator.

Positioning the Cursor

If you use PRINTEXT instructions to prompt the operator for several data items, you would probably want to position the cursor after the first prompt. To position the cursor, you need two instructions: a PRINTEXT instruction and a TERMCTRL instruction:

```
PRINTEXT LINE=1,SPACES=13
TERMCTRL DISPLAY
```

The operands LINE=1 and SPACES=13 cause the system to position the cursor on the fourteenth space of line 1 (the second line). (The lines of a screen are numbered 0 through 23.)

Since the PRINTEXT instruction actually accumulates output in the system buffer, the TERMCTRL instruction is required to cause the cursor to be positioned.

Waiting for a Response

After you issue all the prompts, you must wait for the operator to respond. To wait for a response, use the WAIT instruction as follows:

```
WAIT    KEY
```

The operand KEY means that the program will wait until the operator presses either the enter key or one of the Program Function (PF) keys.

Reading a Data Item

Reading a data item involves issuing a READTEXT instruction for each data item you want to read. The READTEXT instruction might look like this:

```
READTEXT NAME,LINE=1,SPACES=13,MODE=LINE
```

The instruction reads the data item into the storage area called NAME. The operands LINE=1 and SPACES=13 cause the system to look for the data starting in the fourteenth position of the second line of the screen. The operand MODE=LINE allows the data to contain blanks.

Writing a Data Item

Writing a data item means transferring a data item from a storage area to the screen. A PRINTEXT instruction might look like this:

```
PRINTEXT NAME,LINE=11
```

The instruction writes the data item from the storage area called NAME. The operand LINE=11 causes the system to display the data starting in the first position of the twelfth line of the screen.

If you want to display another data item on the next line, you can use the SKIP operand as follows:

```
PRINTEXT ADDR,SKIP=1
```

The SKIP=1 causes the system to skip to the first position of the next line.

To leave spaces between one data item and another, use the SPACES operand as follows:

```
PRINTEXT CITY,SPACES=2
```

The SPACES=2 operand causes the system to leave two blanks between the previous data item and CITY.

Example

Prompt the operator for five data items: name, address, city, state, and zip code. Then display the five data items.

```
1   TEST      PROGRAM   BEG
2   TERM      IOCB      SCREEN=STATIC
3   BEG       ENQT      TERM
4             ERASE     MODE=SCREEN,TYPE=ALL,LINE=0
5             PRINTEXT  '         NAME:',LINE=1,PROTECT=YES
6             PRINTEXT  '      ADDRESS:',SKIP=1,PROTECT=YES
              PRINTEXT  '         CITY:',SKIP=1,PROTECT=YES
              PRINTEXT  '        STATE:',SKIP=1,PROTECT=YES
              PRINTEXT  '          ZIP:',SKIP=1,PROTECT=YES
7             PRINTEXT  LINE=1,SPACES=13
8             TERMCTRL  DISPLAY
9             WAIT      KEY
10            READTEXT  NAME,LINE=1,SPACES=13,MODE=LINE
11            READTEXT  ADDR,LINE=2,SPACES=13,MODE=LINE
              READTEXT  CITY,LINE=3,SPACES=13,MODE=LINE
              READTEXT  ST,LINE=4,SPACES=13
              READTEXT  ZIP,LINE=5,SPACES=13
12            PRINTEXT  NAME,LINE=11
13            PRINTEXT  ADDR,SKIP=1
              PRINTEXT  CITY,SKIP=1
14            PRINTEXT  ST,SPACES=1
              PRINTEXT  ZIP,SPACES=2
15            TERMCTRL  DISPLAY
16            DEQT
              PROGSTOP
    NAME      TEXT      LENGTH=30
    ADDR      TEXT      LENGTH=30
    CITY      TEXT      LENGTH=30
    ST        TEXT      LENGTH=2
    ZIP       TEXT      LENGTH=5
              ENDPROG
              END
```

1 Begin the program and execute the instruction at label BEG.

2 Define the screen as static.

3 Get exclusive use of the terminal.

4 Erase the screen. Erase the entire screen (MODE=SCREEN), including protected and unprotected fields (TYPE=ALL), and begin on the first line of the screen (LINE=0).

5 Prompt the operator for name. Display the prompt on the second line of the screen (LINE=1) and prevent the operator from overlaying the prompt (PROTECT=YES).

6 Prompt the operator for address. Display the prompt one line below the previous prompt (SKIP=1) and prevent the operator from overlaying the prompt (PROTECT=YES).

7 Position the cursor on the fourteenth space (SPACES=13) of the second line of the screen (LINE=1).

8 Cause the cursor to be positioned (the previous PRINTEXT instruction accumulates output in the system buffer).

9 Wait for the operator to respond to the prompts. Resume execution when the operator presses either the enter key or one of the Program Function keys.

10 Read the first data item. Look for the data in the fourteenth space (SPACES=13) of the second line of the screen (LINE=1) and allow blanks in the data (MODE=LINE).

11 Read the second data item (address). Look for the data in the fourteenth space (SPACES=13) of the third line of the screen (LINE=2) and allow blanks in the data (MODE=LINE).

12 Display the data item NAME. Begin displaying the data on the first position of the twelfth line of the screen (LINE=11).

13 Display the data item ADDR. Begin displaying the data on the first position of the next line (SKIP=1). (In this example, ADDR would appear on the thirteenth line of the screen.)

14 Display the data item ST. Begin displaying the data after leaving one space (SPACES=1). (In this example, data item ST would appear one space to the right of data item CITY.)

15 Cause the data in ZIP to be displayed. (The data in ZIP remains in the system buffer until you issue this instruction or end the program with a PROGSTOP.)

16 Relinquish exclusive use of the terminal.

Writing the Screen Image to a Data Set

This section shows how to create a screen image and use it in a program. The approach assumes that you want to write a program that can execute on either a 4978 or 3101 Display Terminal.

For more information on writing a screen image to a data set, see Appendix C, "Static Screens and Device Considerations" on page PG-269.

Writing a screen to a data set and using it in a program requires that you do the following things:

Writing the Screen Image to a Data Set *(continued)*

1. Create the screen.
2. Define the screen as static.
3. Read the screen into a buffer.
4. Get exclusive access to the terminal.
5. Display the screen and position the cursor.
6. Reserve storage for data.
7. Wait for a response.
8. Read a data item.
9. Write a data item.
10. Link-edit the program.

Creating a Screen

To create a screen image, use the $IMAGE utility as follows:

1. From the session manager, select option 4 (TERMINAL UTILITIES) from the primary option menu.
2. Then select option 4 ($IMAGE). This option loads the $IMAGE utility.
3. Define a null character when the COMMAND(?) prompt appears by by entering:

```
COMMAND (?): NULL @
```

   You will use the null character to define unprotected fields. *Unprotected fields* are the fields in which the operator will enter data.

4. Define the screen dimensions as 24 by 80 (full screen) by entering:

```
COMMAND (?): DIMS 24 80
```

5. Enter the command EDIT. A blank screen appears.

6. Press the PF1 key to enter define mode. While in define mode, you can define the screen.

7. Enter the text for your screen image. Enter the fixed part of the screen exactly as you want it to appear on the screen. The fixed fields are called *protected fields*. Use the null character (@) to define the unprotected data fields.

   The screen looks as follows:

```
                                                 (line 0)
   NAME: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          (line 1)
ADDRESS: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          (line 2)
   CITY: @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@          (line 3)
  STATE: @@                                      (line 4)
    ZIP: @@@@@                                   (line 5)
```

8. Press the enter key after you complete the design of your screen image. The enter key takes you out of define mode.

9. Press the PF3 key to return to the $IMAGE command mode.

10. Save your new screen image in data set AP08CSCR on volume EDX002 by entering:

```
SAVE AP08CSCR,EDX002
```

11. In response to the message:

```
SHOULD THE 3101 DATASTREAM BE SAVED?
```

   reply N. (You would reply Y if you coded attributes (such as blinking or nondisplay) that are available on the 3101 Display Terminal.)

At this point, the system saves the screen. Use the EN command to end the $IMAGE utility.

For more information on creating a screen image, refer to *Language Reference*.

Writing the Screen Image to a Data Set *(continued)*

Defining the Screen as Static

To define a screen as static, use the IOCB statement as follows:

```
TERM   IOCB   SCREEN=STATIC,                                         X
              BUFFER=IOBUF,                                          X
              OVFLINE=YES,                                           X
              LEFTM=0,                                               X
              RIGHTM=1919,                                           X
              TOPM=0,                                                X
              BOTM=23
```

This statement defines the loading terminal as a static screen. The label TERM defines the name you will use in other instructions in the program. The BUFFER operand identifies IOBUF as the buffer that will be associated with the screen. The OVFLINE operand tells the system to continue a line that exceeds the right margin on the next line. The next four operands (LEFTM, RIGHTM, TOPM, and BOTM) define the static screen as the entire physical screen (lines 0-23 and columns 0-79).

Note: Remember that to continue a line, the continued line must begin in column 16.

For information on defining logical screens, see Appendix C, “Static Screens and Device Considerations” on page PG-269.

Reading the Screen Image into a Buffer

To read the screen you have created, you need to do the following things:

1. Code the name and volume of the screen in a TEXT statement:

   ```
   DSNAME   TEXT   'AP08CSCR,EDX002'
   ```

   This TEXT statement refers to data set AP08CSCR on volume EDX002. This data set contains the screen you saved when you used the $IMAGE utility.

2. Reserve storage for the screen with a BUFFER statement:

   ```
   DISKBFR BUFFER  1024,BYTES
   ```

   The amount of storage you reserve depends on how many bytes $IMAGE used to store the screen image. For example, if $IMAGE used 900 bytes to store a screen image, use 1024 bytes (the next highest 256-byte increment).

3. Specify the type of image data set you have created:

```
TERMTYPE  DATA  C'4978'
```

   The type of image data set refers to the way you stored the data set. Since you answered **N** to the "SHOULD THE 3101 DATASTREAM BE SAVED?" prompt, the system saved the data set as a 4978 image.

4. Use the CALL instruction to read the screen:

```
CALL   $IMOPEN,(DSNAME),(DISKBFR),(TERMTYPE)
```

   The $IMOPEN subroutine reads the screen from the data set defined by DSNAME and puts the screen into DISKBFR. TERMTYPE refers to the DATA statement that defines the type of image data set.

Getting Exclusive Access to the Terminal

Before you can use a terminal as a static screen, you must get exclusive access to it. Use the ENQT instruction as follows:

```
ENQT  TERM
```

The operand TERM is the name you used to define the terminal in the IOCB instruction.

Displaying the Screen and Positioning the Cursor

Displaying the screen and positioning the cursor involves three instructions.

The first instruction, the CALL $IMPROT instruction, prepares the protected fields for display:

```
CALL   $IMPROT,(DISKBFR),(FTABLE)
```

The presence of the third operand (in this case, FTABLE) causes the instruction to construct what is called a field table. A ***field table*** shows the location and length of each unprotected field on the screen. Define the field table as follows:

```
FTABLE  BUFFER  15,WORDS
```

The field table requires 3 words for each unprotected field.

Writing the Screen Image to a Data Set *(continued)*

The second instruction positions the cursor after the first prompt:

```
PRINTEXT LINE=1,SPACES=9
```

Finally, the third instruction displays the screen:

```
TERMCTRL DISPLAY
```

Reserving Storage for Data

To reserve storage for a data item that you read, you must know its maximum length. To reserve storage for a text string of 5 characters, use the TEXT statement as follows:

```
ZIP      TEXT      LENGTH=5
```

The name of the storage is ZIP. This storage area will eventually contain five bytes of data (the zip code).

Waiting for a Response

After you issue the prompts, you must wait for the operator to respond. To wait for a response, use the WAIT instruction as follows:

```
WAIT    KEY
```

The operand KEY means that the program will wait until the operator presses either the enter key or one of the Program Function (PF) keys.

Reading a Data Item

Reading a data item involves reading all unprotected data from the screen. Use the READTEXT instruction as in the following example:

```
READTEXT IOBUF,MODE=LINE,LINE=0,SPACES=0
```

The instruction reads all unprotected data into the buffer called IOBUF. The operands LINE=0 and SPACES=0 cause the system to look for the data starting in the first position of the screen. MODE=LINE allows for blanks in the input data.

To move each data item into its own storage area, use the following instructions:

```
MOVEA   #1,IOBUF
MOVE    NAME,(0,#1),(30,BYTE)
```

The MOVEA instruction moves the address of buffer containing the unprotected fields. The MOVE instruction moves the 30 bytes at the start of the buffer to NAME.

For each additional field, increment register 1 to the next field in IOBUF and move it to its data area:

```
ADD     #1,FTABLE+4
MOVE    ADDR,(0,#1),(30,BYTE)
```

The ADD instructions adds the size of the first field (NAME) to register 1. The MOVE instruction moves the 30 bytes at IOBUF+30 to ADDR.

Writing the Screen Image to a Data Set *(continued)*

Writing a Data Item

Writing a data item means transferring a data item from a storage area to the screen. A PRINTEXT instruction might look like this:

```
PRINTEXT NAME,LINE=11
```

The instruction writes the data item from the storage area called NAME. The operand LINE=11 causes the system to display the data starting in the first position of the twelfth line of the screen.

If you wanted to display another data item on the next line, you could use the SKIP operand:

```
PRINTEXT CITY,SKIP=1
```

The SKIP=1 causes the system to skip to the first position of the next line before displaying the data item CITY.

To display another data item on the same line, you could use the SPACES operand:

```
PRINTEXT ST,SPACES=1
```

SPACES=1 causes the system to skip one space on the same line before displaying the data item ST. on the same line before displaying the data item ST.

Link-Editing the Program

Using the $IMAGE subroutines ($IMOPEN, $IMDEFN, and $IMPROT) means that you must do one more thing when you link-edit the program. You must reference the $IMAGE subroutines you have used.

You must supply the linkage editor, $EDXLINK, the following "control statements":

```
AUTOCALL $AUTO,ASMLIB
INCLUDE ASMOBJ,EDX002
LINK AP08C,EDX40 REPLACE END
```

The first control statement refers to a library of IBM-supplied routines. Unless you have moved the library, you can code this statement as you see it here.

The second control statement refers to where you put the output of the compiler.

The third control statement says to put the output of the link-edit on volume EDX40, call it AP08C, and replace it if it already exists. END tells $EDXLINK not to expect any other control statements.

You can either create a data set containing these control statements or enter the statements "interactively" each time you execute $EDXLINK.

For more information on link-editing, see Chapter 5, "Preparing an Object Module for Execution" on page PG-81.

Example

Prompt the operator for name, address, city, state, and zip code. Then display the five data items. Use the screen AP08CSCR on volume EDX002 (already defined with the $IMAGE utility).

```
1    TEST       PROGRAM    BEG
2               EXTRN      $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
3    TERM       IOCB       SCREEN=STATIC,                                C
                 BUFFER=IOBUF,OVFLINE=YES,LEFTM=0,                      C
                 RIGHTM=1919,TOPM=0,BOTM=23
4    BEG        CALL       $IMOPEN,(DSNAME),(DISKBFR),(TERMTYPE)
5               MOVE       CODE,TEST+2
6               IF         CODE,NE,-1
                  PRINTEXT 'OPEN ERROR CODE = ',SKIP=1
                  PRINTNUM CODE
                  GOTO      END
                ENDIF
7               ENQT       TERM
8               CALL       $IMPROT,(DISKBFR),(FTABLE)
9               PRINTEXT   LINE=1,SPACES=9
10              TERMCTRL   DISPLAY
11              WAIT       KEY
12              READTEXT   IOBUF,MODE=LINE,LINE=0,SPACES=0
13              MOVEA      #1,IOBUF
14              MOVE       NAME,(0,#1),(30,BYTE)
15              ADD        #1,FTABLE+4
                MOVE       ADDR,(0,#1),(30,BYTE)
                ADD        #1,FTABLE+10
                MOVE       CITY,(0,#1),(30,BYTE)
                ADD        #1,FTABLE+16
                MOVE       ST,(0,#1),(2,BYTE)
                ADD        #1,FTABLE+22
                MOVE       ZIP,(0,#1),(5,BYTE)
16              PRINTEXT   NAME,LINE=11
17              PRINTEXT   ADDR,SKIP=1
                PRINTEXT   CITY,SKIP=1
18              PRINTEXT   ST,SPACES=1
                PRINTEXT   ZIP,SPACES=2
                DEQT
     END        PROGSTOP
19   DSNAME     TEXT       'AP08CSCR,EDX002'
20   DISKBFR    BUFFER     1024,BYTES
21   TERMTYPE   DATA       C'4978'
22   FTABLE     BUFFER     15,WORDS
23   IOBUF      BUFFER     1920,BYTES
     CODE       DC         F'0'
```

```
NAME      TEXT      LENGTH=30
ADDR      TEXT      LENGTH=30
CITY      TEXT      LENGTH=30
ST        TEXT      LENGTH=2
ZIP       TEXT      LENGTH=5
          ENDPROG
          END
```

1 Begin the program and execute the instruction at label BEG.

2 Define as external references the $IMAGE subroutines that the program uses. The linkage editor resolves these external references when you use the autocall option.

3 Define the screen as static.

4 Read the screen from the data set defined by DSNAME. Put the screen in the buffer defined by DISKBFR.

5 Move the return code that resulted from the $IMOPEN subroutine to CODE.

6 If the return code that resulted from the $IMOPEN subroutine does not indicate "successful completion," display an error message and end the program.

7 Get exclusive use of the terminal.

8 Prepare the protected fields for display.

9 Position the cursor on the tenth space (SPACES=9) of the second line of the screen (LINE=1).

10 Display the screen.

11 Wait for the operator to respond to the prompts. Resume execution when the operator presses either the enter key or one of the Program Function keys.

12 Read all unprotected data. Look for the data in the first space (SPACES=0) of the first line of the screen (LINE=0) and allow blanks in the data (MODE=LINE).

13 Move the address of the buffer (IOBUF) that contains the unprotected data into register 1.

14 Move the first 30 characters from the buffer to NAME.

15 Increment register 1 to point to the next data item (address).

16 Display the data item NAME. Begin displaying the data on the first position of the twelvth line of the screen (LINE=11).

Writing the Screen Image to a Data Set *(continued)*

17 Display the data item ADDR. Begin displaying the data on the first position of the next line (SKIP=1). (In this example, ADDR would appear on the thirteenth line of the screen.)

18 Display the data item ST. Begin displaying the data after leaving one space (SPACES=1). (In this example, data item ST would appear one space to the right of data item CITY.)

19 Point to the data set (AP08CSCR on volume EDX002) that contains the screen created with the $IMAGE utility.

20 Reserve storage for the screen. (Except for screens much larger than the one in this example, 1024 bytes is enough storage.)

21 Define the type of image data set to be read. (Coding C'4978' allows you to read the screen to a 4978 or a 3101, whether or not you saved the 3101 datastream. C'3101' allows you to read the screen to a 3101 if you saved the 3101 datastream. If you code C'   ', you can read the screen to a 4978 or 3101 if you saved the 3101 datastream.)

22 Reserve storage for the field table.

23 Reserve storage for the unprotected data.

Chapter 9. Designing Programs

This chapter discusses designing EDL programs.

All of the programs shown so far have had one thing in common: they are all short, self-contained groups of instructions that perform a simple function without interacting with any other program.

This chapter:

- Defines the terms *program* and *task* and describes how to create a program that consists of more than one task
- Describes how to use the same group of instructions from more than one program
- Shows how to use the same storage more than once for different parts of a program (overlays)
- Shows how to improve performance by using storage as a buffer area

What Is a Task?

A *task* is a unit of work that you form by combining instructions. In its simplest form, a task consists of a TASK statement, instructions, and an ENDTASK statement.

Each task runs independently, competing equally with other tasks for system resources.

Designing Programs

What Is a Task? *(continued)*

When you code a task, you assign a priority to the task. A *priority* is a number that determines the rank of the task. The supervisor uses priority to determine which task receives system resources. The highest priority is 1 and the lowest is 510.

In the following example, TASK01 is the name of a task. START01 is the label on the first instruction to be executed, and 140 is the priority of the task.

```
TASK01  TASK  START01,140
        .
        .
        .
        ENDTASK
```

The supervisor places each task in one of five states:

Inactive	Task is detached or is not yet attached
Waiting	Task is waiting for the occurrence of an event or the availability of a resource
Ready	Task is ready but is not the highest priority task
Active	Task is attached and is the highest priority task on its level
Executing	Task is using the processor

Only one task may be active on each of four machine hardware levels. (The supervisor executes on hardware level 1; application programs usually execute on hardware level 2 or 3.)

The active task in each hardware level is the read task that has the highest priority and is not waiting for an event or a resource.

Initiating a Task

You can initiate a task either by loading or attaching it. The system places the primary task in the ready state when you load the program. You can initiate a secondary task with the ATTACH statement if the task is not already active *and* you do either of the following:

- You write a program that consists of a primary task and a secondary task.
- You link-edit a primary task with another task. (You must code an EXTRN statement in the primary task and an ENTRY statement in the secondary task.)

You return a task to the inactive state when you execute either a DETACH instruction or ENDTASK instruction. The DETACH instruction suspends the task and allows it to be attached again.

Only one copy of a task may be active at a time. A task in processor storage remains until you execute an ENDPROG statement in the associated primary task.

What Is a Program?

A ***program*** is a disk- or diskette-resident collection of one or more tasks that can be loaded into storage for execution. Although program and task are sometimes used synonymously (when a program contains a single task), the basic ***executable*** unit is the task; a program is the unit that the system loads into storage.

You can divide a program into two or more tasks if, for example, you need to synchronize execution between the tasks. Another reason to divide a program into tasks is to have more than one task active at the same time.

The name of a program is the name of the data set in which the program resides. A program can be brought into storage either by a terminal operator, a program, or a supervisor program such as the job stream processor. It can be loaded more than once, either in the same partition or in a different partition.

Creating a Single-Task Program

Most applications consist of a single task in a single program. The program contains no execution overlay. The task competes for system resources with other tasks currently in the system.

The following example shows the structure of a single-task program:

```
BEGIN   PROGRAM  START
        .
        .
        .
        PROGSTOP
        ENDPROG
        END
```

In this example, BEGIN is the name of the task, and START is the label of the first instruction to be executed.

Note that even though the TASK statement is not required in a simple program, the program still consists of a single task.

Creating a Single-Task Program *(continued)*

Figure 1 is an example of a single-task program structure.

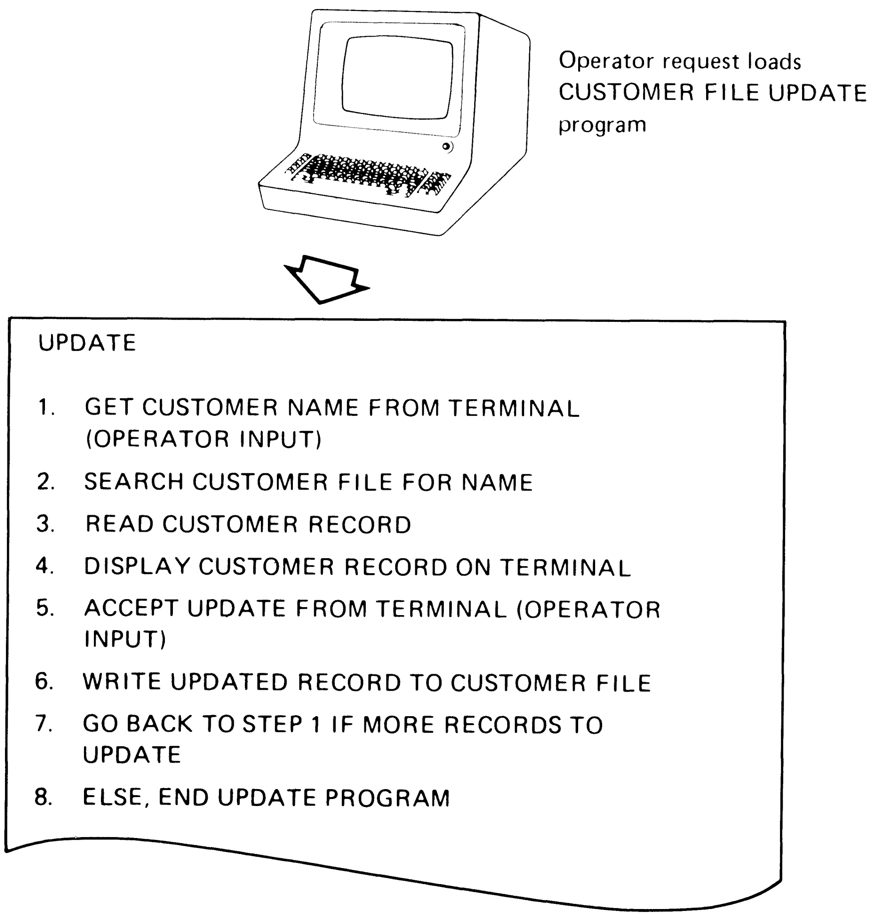


Figure 1. Single-Task Application Example

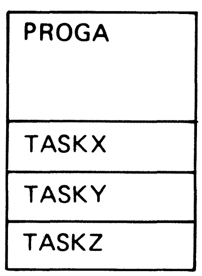
Creating a Multitask Program

A multitask program contains more than one task. For example:

```
BEGIN   PROGRAM   START
        .
        .
        .
        ATTACH    CALC
        .
        .
        .
        PROGSTOP
CALC    TASK
        instructions
        ENDTASK
        ENDPROG
        END
```

Note that the PROGRAM and PROGSTOP statements define a task called the *primary task*. The TASK and ENDTASK statements define a *secondary task*, invoked by the ATTACH instruction.

Figure 2 illustrates multitasking in a single program. When you load the program, the system loads PROGA, called the primary task. The other tasks shown in PROGA start when an active task issues a command (such as an ATTACH instruction) that tells the tasks to begin.



Program made up of multiple tasks
- Concurrent (asynchronous) execution of tasks within program
- Tasks compete for system resources with all other tasks currently in system

Figure 2. Multitask Program Structure

Once in execution, all tasks within a program compete with one another and with all other tasks active in the system. The supervisor considers each task as a discrete unit of work and assigns processor time based on task priority, regardless of whether a task is the primary task of a program. All tasks compete for resources based on assigned priorities.

If a primary task ends before the secondary task, the secondary task runs to completion.

Synchronizing Tasks

You can synchronize tasks with the WAIT and POST instructions or with the DETACH and ATTACH instructions. If you use the WAIT and POST instructions, the waiting task must contain an event control block (ECB) that can be posted by the POST instruction. Execution then continues in the waiting task at the first instruction after the WAIT instruction. A task can also wait for the operator to press a Program Function (PF) key, for a time interval to occur, or for a program to finish execution.

While waiting to be posted, the task enters a waiting state. The task also enters a waiting state if it is waiting for a read or write operation to occur or if it has executed a DETACH instruction.

You can use the DETACH and ATTACH instruction to synchronize tasks the same way you use the WAIT and POST instruction, with the following differences:

- The attached task becomes enqueued to the currently active terminal for the task that issued the ATTACH instruction.
- The system provides the ECB.
- You cannot use the ATTACH and DETACH instructions from within subroutines.

Defining and Calling Subroutines

In a program, certain functions may need to be repeated at different points in a program. For example, you do not need to code the same sequence of instructions each time your program needs to perform a given arithmetic function. You can code the instructions once and define them as a subroutine. You can than enter and execute that subroutine from as many points in your program as needed. You can also use the subroutine in another program by including it at link-edit time.

The following instructions provide the means for defining and calling subroutines:

CALL Transfers control to a subroutine

RETURN Returns control from the subroutine to the calling program

SUBROUT Defines the entry point and parameters of a subroutine

EXTRN Defines an external reference

ENTRY Defines a program entry point

Defining a Subroutine

Use SUBROUT to define the entry point of a subroutine. You can specify up to five parameters as arguments in the subroutine. The subroutine must include a RETURN instruction to provide linkage back to the calling task. You can have nested subroutines, and a maximum of 99 subroutines are permitted per program. If you assemble your subroutine as an object module that can be link-edited, you must code an ENTRY statement 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.

The syntax of the SUBROUT instruction is as follows:

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

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

Code the *name* operand with the symbolic name of the subroutine to be referred to by other instructions. The *label* field is optional. Do not confuse the *label* field with the subroutine name you specify in the *name* operand.

Passing Parameters in a Subroutine (Example)

Par1 through *par5* are the parameter names to be passed to the subroutine when it is entered. These names must be unique to the whole program. All parameters defined outside the subroutine are known within the subroutine. Thus, you need to define only parameters that may vary with each call to a subroutine.

For instance, assume two calls to the same subroutine. The first call passes parameters A and C and the second CALL passes parameters B and C. Because C is common to both, you need not define it in the SUBROUT instruction.

In the following example, a program calls subroutine CHKBUFF, passing two parameters. The first (BUFFLEN) is a variable containing the maximum allowable buffer count. The second (BUFFEND) is the address of the instruction to be executed if the buffer is full.

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

Defining and Calling Subroutines *(continued)*

Calling a Subroutine

Use the CALL instruction to execute your subroutine.

If the called subroutine is a separate object module to be link-edited with your program, then you must code an EXTRN statement for the subroutine name in the calling program.

The syntax of the CALL instruction is as follows:

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

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

The *name* operand is the name of the subroutine to be executed.

Par1 through *par5* are the parameters associated with the subroutine. You can pass Up to five single-precision integers, labels of single-precision integers, or null parameters to the subroutine. The actual constant or the value at the named location moves to the corresponding subroutine parameter.

If you enclose the parameter name in parentheses, the address of the variable passes to the subroutine. The address can be the label of the first word of any type of data item or data array. Within the subroutine, you must move the passed address of the data item into index registers #1 or #2 to reference the data item. If the parameter name enclosed in parentheses is a symbol defined by an EQU instruction, the system passes the value of the symbol.

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

Subroutine Call Examples

The following example passes the value 5 to the subroutine PROG:

```
CALL   PROG,5
```

The following example passes the value 5 and the null parameter 0 to the subroutine CALC:

```
CALL   CALC,5,
```

The following example passes the contents of PARM1, the address of PARM2, and the value of the EQU symbol FIVE:

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

Defining and Calling Subroutines *(continued)*

Calling a Subroutine Passing Integer Parameters (Example)

The following example shows a program that passes integers to a subroutine:

```
SUBEXAMP         PROGRAM          START
START            CALL             CALC,50,SUM1
                  .
                  .
                  .
C2               CALL             CALC,SUM1,SUM2
                  .
                  .
                  .
                 PROGSTOP
INTEGERA         DATA              F'10'
INTEGERB         DATA              F'15'
SUM1             DATA              F'0'
SUM2             DATA              F'0'
SUB1             SUBROUT           CALC,XVAL,YVAL
A1               ADD               INTEGERA,XVAL,RESULT=YVAL
                 RETURN
                 ENDPROG
                 END
```

In the first CALL, the first parameter (the integer value 50) corresponds to the first parameter defined in the subroutine (XVAL). Program location SUM1 corresponds to the second parameter (YVAL). When the ADD instruction executes, the system substitutes 50 for XVAL and location SUM1 for YVAL. After the ADD instruction, SUM1 equals 60, the sum of INTEGERA and 50.

The second call causes 70, the sum of SUM1 and INTEGERA, to be put in location SUM2. Because INTEGERA does not change, you do not need to pass it as a parameter.

Reusing Storage using Overlays

You can reuse a single storage area allocated to a program by using overlays. EDL provides two kinds of overlays: overlay segments and overlay programs.

An *overlay segment* is a self-contained portion of a program that is called and executed as a synchronous task. The program that calls the overlay segment need not be in storage while the overlay segment is executing. Overlay segments perform a specific function and generally execute only once.

An *overlay program* is a self-contained portion of a program that is loaded and executed as an asynchronous task. Overlay programs require a main *control program* that controls the execution of up to nine overlay programs.

Using Overlay Segments

Figure 3 shows the structure of an application program that is split into a root segment and three overlay segments. When you load the main program, the loader reserves enough space for the root segment, the overlay area manager, the overlay control table, and the largest overlay segment as shown in Figure 4 .

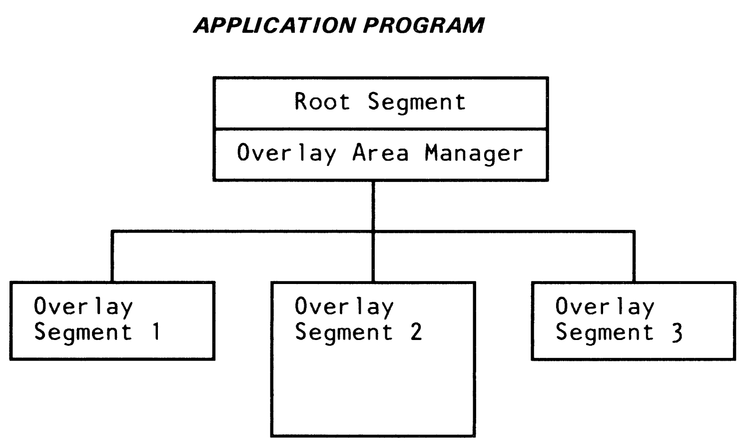


Figure 3. Application Overlay Segments

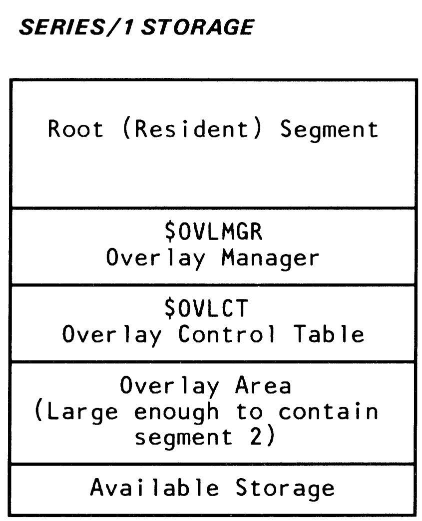


Figure 4. Overlay Segments in Series/1 Storage

The following example shows a root segment and three overlay segments:

```
BEGIN   PROGRAM   START
        EXTRN     CALC,UPDATE,WRITE
        .
        .
        CALL      CALC
        .
        .
        CALL      UPDATE
        .
        .
        CALL      WRITE
        .
        .
        PROGSTOP
        ENDPROG
        END
*****************************************************************
*       OVERLAY SEGMENT 1                                       *
*****************************************************************
        SUBROUT   CALC
        ENTRY     CALC
          instructions
        RETURN
        END
*****************************************************************
*       OVERLAY SEGMENT 2                                       *
*****************************************************************
        SUBROUT   UPDATE
        ENTRY     UPDATE
          instructions
        RETURN
        END
*****************************************************************
*       OVERLAY SEGMENT 3                                       *
*****************************************************************
        SUBROUT   WRITE
        ENTRY     WRITE
          instructions
        RETURN
        END
```

Each of the overlay segments is a subroutine that you can compile separately.

Creating an Overlay Structure

To create an overlay structure, use the linkage editor $EDXLINK. The linkage editor allows you to combine the overlay segments you link-edited separately into a program segment overlay structure. $EDXLINK automatically includes an overlay manager with the root segment, along with an overlay area equal to the largest overlay segment. A CALL (or transfer of control) to a module within an overlay segment triggers the overlay area manager to load the overlay segment into the overlay area and transfer control to it. Overlay segments execute as synchronous tasks. An overlay segment cannot call another overlay segment.

Overlay segments are specified in the OVERLAY statement of $EDXLINK which is discussed in detail in Chapter 5, "Preparing an Object Module for Execution" on page PG-81.

Reusing Storage using Overlays *(continued)*

Overlay Programs

An *overlay program* is a program in which certain control sections can use the same storage location at different times during execution. Overlay programs execute concurrently as asynchronous tasks with other programs and are specified in the PROGRAM statement in the main program.

With overlay programs, the main program loads the overlay programs. The loader allocates the overlay area for overlay programs at main program load time. The overlay area is equal to the largest overlay program listed in the main program header.

In Figure 5, the application is split into separate programs. PHASE1, the primary program, loads the overlay programs (PHASE2, PHASE3, and PHASE4) as requested. When PHASE1 is loaded, the loader recognizes that overlay programs are referenced. The loader looks at each overlay program and reserves enough storage to hold PHASE1 plus the largest overlay program (PHASE3) as shown in Figure 6 on page PG-151.

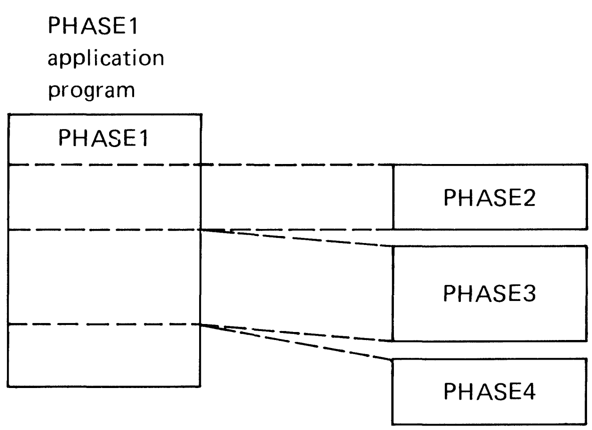


Figure 5. EDL Overlay Programs

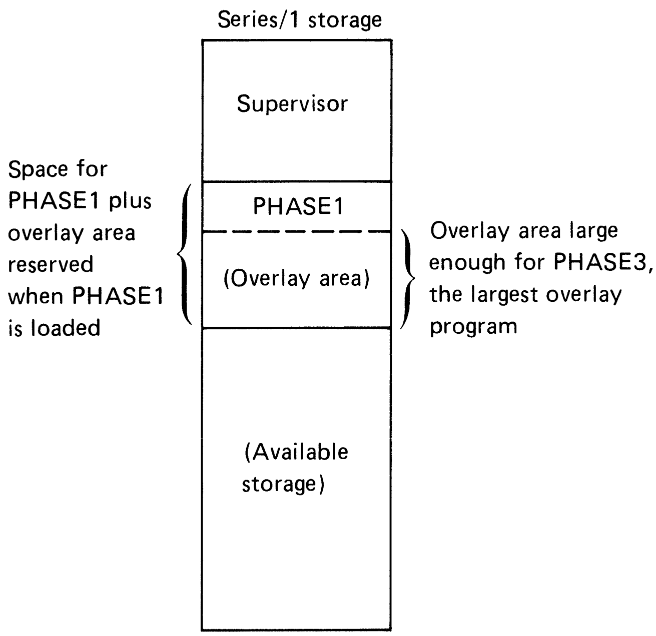


Figure 6. EDL Overlay Programs in Series/1 Storage

As each overlay program completes execution, PHASE1 loads the next overlay program, until all required programs have run. When PHASE1 terminates, the system releases the storage reserved for PHASE1 and its overlay programs. See the *Language Reference* for information on coding the PROGRAM statement for overlays.

Using Large Amounts of Storage (Unmapped Storage)

Unmapped storage allows you to write a program that uses large amounts of storage. Unmapped storage allows you to store large amounts of data and retrieve data faster than you could retrieve it from disk or diskette. This section describes setting up, obtaining, accessing, and releasing unmapped storage.

What Is Unmapped Storage?

Unmapped storage is storage that has not been reserved by the SYSTEM statement.

Setting up Unmapped Storage

Use the STORBLK statement to define the size and number of the unmapped storage areas a program will use. The TWOKBLK operand defines the size of each unmapped storage area. For example, if you need unmapped storage areas to accommodate 6000 bytes of data, code TWOKBLK=3 (6K = 6144 bytes). The maximum size of an unmapped storage area is 65,536 bytes (TWOKBLK=32).

The MAX operand defines the number of unmapped storage areas. For example, if you need ten unmapped storage areas, code MAX=10.

In the following example, HOLD defines 16 (MAX=16) 2K-byte areas of unmapped storage.

```
HOLD   STORBLK   TWOKBLK=1,MAX=16
```

The STORBLK statement also sets up a mapped storage area the same size as the unmapped storage area.

Obtaining Unmapped Storage

Use the GETSTG instruction to obtain the mapped and unmapped storage areas you defined in the STORBLK statement. For example:

```
         GETSTG   HOLD,TYPE=ALL
```

This instruction obtains the mapped and unmapped storage that you defined in the STORBLK statement with the label HOLD. The size of the area depends on the TWOKBLK operand of the STORBLK statement. The operand TYPE=ALL tells the system to obtain the unmapped and mapped storage areas. The number of unmapped storage areas the system obtains depends on the MAX parameter of the STORBLK statement.

If you want to obtain only one unmapped storage area, code the GETSTG instruction as follows:

```
    GETSTG    HOLD,TYPE=NEXT
```

The instruction causes the system to obtain an unmapped storage area that you defined in the STORBLK statement with the label HOLD. The size of the area depends on the TWOKBLK operand of the STORBLK statement. The system obtains one unmapped storage area. For example, if you specified MAX=24 on the STORBLK statement and the system had already obtained fifteen unmapped storage areas, the system would obtain the sixteenth one.

Using an Unmapped Storage Area

You can use an unmapped storage area just like you would use any other storage area. For example, you can move data into the area or perform calculations on data within the area.

The SWAP instruction allows you to use an unmapped storage area. For example:

```
SWAP    HOLD,USANO
```

The instruction allows you to access the unmapped storage area defined by the STORBLK statement at label HOLD. The operand USANO refers to the label of a DATA statement that defines the number of the unmapped storage area you want to access. For example, if USANO contains "5", the SWAP instruction allows the program to access the fifth unmapped storage area.

You can also code the number of the unmapped storage area you want to use:

```
SWAP    HOLD,10
```

This instruction allows you to use the tenth unmapped storage area defined by the STORBLK statement at label HOLD. Until you execute another SWAP instruction, you can use only the tenth unmapped storage area.

Notes:

1. You can use only one unmapped storage area at a time.

2. While you are using an unmapped storage area, you cannot use the mapped storage area.

Releasing Unmapped Storage

Use the FREESTG instruction to release any unmapped storage area that you obtained with the GETSTG instruction. For example:

```
FREESTG   HOLD,TYPE=ALL
```

This instruction releases the unmapped storage areas defined by the STORBLK statement at label HOLD. The operand TYPE=ALL causes the instruction to release all of the storage areas. For example, if the STORBLK statement specifies MAX=16, this instruction causes all sixteen unmapped storage areas and the mapped storage area to be released.

Using Large Amounts of Storage (Unmapped Storage) *(continued)*

Example

The following example uses sixteen unmapped storage areas, one for each country in South America, to create a table of actuarial data. The table for each of the sixteen countries consists of three-digit mortality rates. The program accumulates 100 rates for both men and women. The unmapped storage the program uses is determined by the country number.

The input records have the following format:

```
Country number      2 bytes
Sex code            1 byte
Age                 2 bytes
Death rate          3 bytes
```

The program:

```
     INSURE   PROGRAM  ST,DS=((ACTTAB,EDX40))
1             COPY     STOREQU
     ST       EQU      *
2             GETSTG   HOLD,TYPE=ALL
                (initialize storage areas)
3    READ     READ     DS1,MORTAL,1,END=STOP
4             MOVE     USANO,CNTRY,(2,BYTES)
5             MOVE     #1,HOLD+$STORMAP
6             SWAP     HOLD,USANO
7             MOVE     #2,AGE
8             MULT     #2,3
9             ADD      #1,#2
10            IF       (SEX,EQ,1)
11              MOVE     (+MENTBL,#1),RATE
              ELSE
                MOVE     (+WMNTBL,#1),RATE
              ENDIF
              GOTO     READ
     STOP     EQU      *
                 (save the unmapped storage areas)
              PROGSTOP
     USANO    DATA     F'0'
12   HOLD     STORBLK  TWOKBLK=1,MAX=16
     MENTBL   EQU      0
     WMNTBL   EQU      MENTBL+300
     MORTAL   BUFFER   256,BYTES
     CNTRY    EQU      MORTAL
     SEX      EQU      MORTAL+2
     AGE      EQU      MORTAL+3
     RATE     EQU      MORTAL+5
              ENDPROG
              END
```

1 Copy the storage control block equates into the program.

2 Obtain the mapped and unmapped storage (one 2K-byte mapped storage area and sixteen 2K-byte unmapped storage areas) specified in the STORBLK statement with the label HOLD.

3 Read an input record from data set ACTTAB on volume EDX40 into the buffer with the label MORTAL.

4 Move the country number in the input record to USANO.

5 Move the address of the mapped storage area into register 1.

6 Use the country number to access the appropriate unmapped storage area.

7 Move the age into register 2.

8 Multiply the age by 3 to arrive at the proper offset into the table.

9 Add the offset to the address of the mapped storage area.

10 Test the sex code for 1 (1 = men).

11 Move the mortality rate into the appropriate slot in the MENTBL (the men's mortality rate table).

12 Set up a 2K-byte mapped storage area and sixteen 2K-byte unmapped storage areas.

Notes

Chapter 10. Performing Data Management from a Program

This section describes ways to accomplish data management from a program. Topics discussed are:

- Allocating, deleting, and renaming a data set
- Opening a data set
- Setting logical end of file
- Finding the device type

To perform other data management functions from an application program such as allocating, deleting, and renaming volumes, see Chapter 13, "Communicating with Other Programs (Virtual Terminals)" on page PG-215.

Performing Data Management from a Program

The $DISKUT3 program enables you to perform the following data management operations from a program:

- Allocate a data set.
- Open a data set.

Performing Data Management from a Program *(continued)*

- Delete a data set.
- Release unused space in a data set.
- Rename a data set.
- Set end-of-data on a data set.

$DISKUT3 allows you to open and set end-of-data on disk, diskette, or tape data sets. You can perform the other four operations (allocating, deleting, releasing unused space, and renaming) on disk or diskette data sets only.

For more information on $DISKUT3, including a list of return codes, refer to *Language Reference*.

You might use $DISKUT3 for any of the following reasons:

- Your program requires more than nine data sets.
- You do not know, at the time you load a program, whether or not the program will need a data set.
- You need to perform several data management functions in one program.
- You want the processor storage that $DISKUT3 requires to be available when $DISKUT3 finishes executing.

To use $DISKUT3, you should be aware of the following factors:

- $DISKUT3 requires about 6.25K bytes of processor storage.
- If you need only to open a data set, $DISKUT3 will be slower than DSOPEN.
- You need to perform error recovery if the system cannot load $DISKUT3.

Allocating a Data Set

The following example shows how to allocate a data set from an application program:

```
     TASK      PROGRAM GO
     GO        EQU     *
               .
               .
1              LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2              WAIT    DSK3EVNT
               .
               .
               PROGSTOP
```

```
3   DSK3EVNT ECB    0
4   LISTPTR1 DC     A(LIST1)
5   LIST1    DC     A(REQUEST1)
6            DC     F'0'
7   REQUEST1 DC     F'2'
8            DC     A(DSX)
9            DC     D'50'
10           DC     F'1'
11           DSCB   DS#=DSX,DSNAME=DATA4
12           COPY     DSCBEQU
             ENDPROG
             END
```

1 Load $DISKUT3 to allocate data set DATA4. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific allocate request.

6 Indicate the end of the list of requests.

7 Request an allocate (2).

8 Point to the DSCB for the data set to be allocated. (The allocate function requires that the data set being allocated be defined by a DSCB.)

9 Indicate that 50 records are to be allocated.

10 Indicate that the data set type is *data*.

11 Define a DSCB for the data set to be allocated.

12 Copy the DSCB equates into the program.

If you attempt to allocate a data set that already exists, $DISKUT3 considers the operation successful if:

- The type of the data set that already exists matches the type on the data set you are allocating
- The size of the data set that already exists matches the size of the data set you are allocating

Performing Data Management from a Program *(continued)*

Opening a Data Set

If you have defined a data set with a DSCB, you need to open the data set from your application program.

The following example shows how to open a data set from an application program:

```
          TASK      PROGRAM GO
          GO        EQU     *
                    .
                    .
1                   LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2                   WAIT    DSK3EVNT
                    .
                    .
3         DSK3EVNT  ECB    0
4         LISTPTR1  DC     A(LIST1)
5         LIST1     DC     A(REQUEST1)
6                   DC     F'0'
7         REQUEST1  DC     F'1'
8                   DC     A(DSY)
9                   DC     D'0'
10                  DC     F'-1'
11                  DSCB   DS#=DSY,DSNAME=DATA4
12                  COPY    DSCBEQU
                    ENDPROG
                    END
```

1 Load $DISKUT3 to open data set DATA4. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific open request.

6 Indicate the end of the list of requests.

7 Request an open (1).

8 Point to the DSCB for the data set to be opened.

9 This doubleword is not used for an open request.

10 Tell $DISKUT3 to return the type of the data set being opened (0 for undefined, 1 for data, 3 for program).

11 Define a DSCB for the data set to be opened.

12 Copy the DSCB equates into the program.

Performing Data Management from a Program *(continued)*

Deleting a Data Set

The following example shows how to delete a data set from an application program:

```
        TASK      PROGRAM GO,DS=((MASTER,EDX002),(UPDATE,EDX003))
        GO        EQU     *
                  .
                  .
1                 LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2                 WAIT    DSK3EVNT
                  .
                  .
3       DSK3EVNT  ECB     0
4       LISTPTR1  DC      A(LIST1)
5       LIST1     DC      A(REQUEST1)
6                 DC      F'0'
7       REQUEST1  DC      F'4'
8                 DC      A(DS2)
9                 DC      D'0'
10                DC      F'-1'
11                COPY    DSCBEQU
                  ENDPROG
                  END
```

1 Load $DISKUT3 to detete data set UPDATE on volume EDX003. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific delete request.

6 Indicate the end of the list of requests.

7 Request a delete (4).

8 Point to the DSCB for the data set to be deleted (UPDATE on volume (EDX003).

9 This doubleword is not used for an delete request.

10 Tell $DISKUT3 to return the type of the data set being deleted (0 for undefined, 1 for data, 3 for program).

11 Copy the DSCB equates into the program.

If you try to delete a data set that does not exist, $DISKUT3 considers the operation to be successful.

Performing Data Management from a Program *(continued)*

Releasing Unused Space in a Data Set

The following example shows how to release unused space in a data set from an application program:

```
        TASK      PROGRAM GO
        GO        EQU     *
                  .
                  .
1                 LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2                 WAIT    DSK3EVNT
                  .
                  .
3       DSK3EVNT  ECB     0
4       LISTPTR1  DC      A(LIST1)
5       LIST1     DC      A(REQUEST1)
6                 DC      F'0'
7       REQUEST1  DC      F'5'
8                 DC      A(DSX)
9                 DC      D'0'
10                DC      F'-1'
11                DSCB    DS#=DSX,DSNAME=TRANS
12                COPY    DSCBEQU
                  ENDPROG
                  END
```

1 Load $DISKUT3 to release space on data set TRANS. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific release request.

6 Indicate the end of the list of requests.

7 Request a release (5).

8 Point to the DSCB for the data set on which space to be released (TRANS).

9 This doubleword is not used for a release request.

10 Tell $DISKUT3 to return the type of the data set on which space is being released (0 for undefined, 1 for data, 3 for program).

11 Define a DSCB for the data set on which to release unused space.

12 Copy the DSCB equates into the program.

Performing Data Management from a Program *(continued)*

Renaming a Data Set

The following example shows how to rename in a data set from an application program:

```
   TASK      PROGRAM GO,DS=((MASTER,EDX003))
   GO        EQU     *
             .
             .
1            LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2            WAIT    DSK3EVNT
             .
             .
3  DSK3EVNT  ECB     0
4  LISTPTR1  DC      A(LIST1)
5  LIST1     DC      A(REQUEST1)
6            DC      F'0'
7  REQUEST1  DC      F'3'
8            DC      A(DS1)
9            DC      F'0'
10           DC      A(NEWNAME)
11           DC      F'-1'
12           COPY    DSCBEQU
13 NEWNAME   DC      CL8'NEWMAST '
             ENDPROG
             END
```

1 Load $DISKUT3 to rename data set MASTER. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific rename request.

6 Indicate the end of the list of requests.

7 Request a rename (3).

8 Point to the DSCB for the data set to be renamed (MASTER on volume EDX003).

9 This word is not used for a rename request.

10 Point to the new data set name.

11 Tell $DISKUT3 to return the type of the data set being renamed (0 for undefined, 1 for data, 3 for program).

12 Copy the DSCB equates into the program.

13 Define the new name for the data set.

Performing Data Management from a Program *(continued)*

Setting End-of-Data on a Data Set

If you define a data set with a DSCB, you need to set end-of-data from your application program.

The following example shows how to set end-of-data on a data set from an application program:

```
     TASK      PROGRAM GO,DS=((MASTER,EDX003))
     GO        EQU     *
               .
               .
1              LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2              WAIT    DSK3EVNT
               .
               .
3    DSK3EVNT  ECB     0
4    LISTPTR1  DC      A(LIST1)
5    LIST1     DC      A(REQUEST1)
6              DC      F'0'
7    REQUEST1  DC      F'6'
8              DC      A(DS1)
9              DC      D'0'
10             DC      F'-1'
11             COPY    DSCBEQU
               ENDPROG
               END
```

1 Load $DISKUT3 to set end-of-data on data set MASTER. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the specific end-of-data request.

6 Indicate the end of the list of requests.

7 Request end-of-data (6).

8 Point to the DSCB for the data set on which to set end-of-data (MASTER on volume EDX003).

9 Indicate that the last record is full. (If the last record is not yet full, this field would contain the number of bytes in the last record.)

10 Tell $DISKUT3 to return the type of the data set on which end-of-data is being set (0 for undefined, 1 for data, 3 for program).

11 Copy the DSCB equates into the program.

Performing Data Management from a Program *(continued)*

Performing More Than One Operation at Once

$DISKUT3 allows you to perform more than one operation with one invocation of the program. For example, you can delete two data sets and allocate a third without loading $DISKUT3 more than once.

The following example shows how to delete two data sets and allocate one data set:

```
    TASK      PROGRAM GO,DS=((MASTER,EDX003),(UPDATE,EDX002))
    GO        EQU     *
              .
              .
1             LOAD    $DISKUT3,LISTPTR1,EVENT=DSK3EVNT
2             WAIT    DSK3EVNT
              .
              .
3   DSK3EVNT  ECB     0
4   LISTPTR1  DC      A(LIST1)
5   LIST1     DC      A(REQUEST1)
6             DC      A(REQUEST2)
7             DC      A(REQUEST3)
8             DC      F'0'
9   REQUEST1  DC      F'4'
10            DC      A(DS1)
11            DC      D'0'
12            DC      F'-1'
13  REQUEST2  DC      F'4'
14            DC      A(DS2)
              DC      D'0'
              DC      F'-1'
15  REQUEST3  DC      F'2'
16            DC      A(DSA)
17            DC      D'300'
18            DC      F'1'
19            COPY    DSCBEQU
20            DSCB    DS#=DSA,DSNAME=NEWMAST,VOLSER=EDX003
              ENDPROG
              END
```

1 Load $DISKUT3 to delete data sets MASTER and UPDATE and to allocate data set NEWMAST. Specify the address (LISTPTR1) of the list of requests (in this case, a single request). Identify the event (EVENT=DSK3EVNT) to be posted when $DISKUT3 completes.

2 Wait for the system to indicate the end of $DISKUT3 by posting DSK3EVNT.

3 Set the initial state of the event control block to zero.

4 Point to the list of requests at LIST1.

5 Point to the request to delete data set MASTER.

6 Point to the request to delete data set UPDATE.

7 Point to the request to allocate data set NEWMAST.

8 Indicate the end of the list of requests.

9 Request a delete (4).

10 Point to the DSCB for the first data set to be deleted (MASTER on volume EDX003).

11 This doubleword is not used for delete requests.

12 Tell $DISKUT3 to return the type of the data set being deleted (0 for undefined, 1 for data, 3 for program).

13 Request a delete (4).

14 Point to the DSCB for the second data set to be deleted (UPDATE on volume EDX002).

15 Request an allocate (2).

16 Point to the DSCB for the data set to be allocated (NEWMAST).

17 Allocate 300 records.

18 Indicate that the data set type is *data*.

19 Copy the DSCB equates into the program.

20 Define a DSCB for the data set being allocated (NEWMAST on volume EDX003).

Opening a Data Set (DSOPEN)

You can open a disk, diskette, or tape data set from a program with the DSOPEN copy code. DSOPEN does the same thing that the system does when you specify a data set in the PROGRAM statement and load the program with either the $L operator command or the LOAD instruction.

Note: Only one DSCB can be open to a tape at a time. If you open a tape data set, you must close the data set before you can open another tape data set.

You might use DSOPEN for any of the following reasons:

- Your program requires more than nine data sets.
- You do not know, at the time you load a program, whether or not the program will need a data set.
- You need to open a data set and do not want to load $DISKUT3 (the system does not need to load DSOPEN).
- The processor storage that $DISKUT3 requires is not available (DSOPEN requires about 1.5K bytes).

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   TCBEQU
COPY   PROGEQU
COPY   DDBEQU
COPY   DSCBEQU
COPY   DSOPEN
```

Note: You must code the equates in the order given.

During execution, invoke DSOPEN with the CALL instruction as follows:

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

Error Exits

If an error occurs while DSOPEN executes, the system transfers control to one of several error exit routines. You must define these routines in your program and move their addresses to labels that are contained in DSOPEN before you call DSOPEN. The routines cannot be subroutines.

The labels and their meanings are as follows:

Label	***Description***
$DSNFND	Data set name not found in directory. If DSOPEN cannot 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. (For more information, refer to the *Language Reference* more information. See the READ instruction return codes for more information.
$$EXIT	Exit address. If $$EXIT is 0 and $DSCBNAME is $$ or $$EDXVOL, DSOPEN initializes the DSCB to the first record (first record in the library) of the volume specified in the $DSCBVOL. If $$EXIT is 0 and $DSCBNAME is $$EDXVOL, DSOPEN initializes the DSCB to the first record of the device where the volume specified on $DSCBVOL resides.
$DSDCEA	Address of an area for DSOPEN to store the directory control entry (DCE). This label contains a 0 if this area does not exist.

If you define an error exit routine as a word of zeroes or move a zero to one of the labels, DSOPEN transfers control to the next sequential instruction after the CALL instruction. For example, the following instruction causes control to return to the next sequential instruction if DSOPEN cannot find the data set:

```
         MOVEA  $DSNFND,LIBEXIT
         .
         .
LIBEXIT  DATA   F'0'
```

The following instruction causes control to return to the next sequential instruction if DSOPEN cannot the volume:

```
         MOVEA  $DSBVOL,0
```

DSOPEN Considerations

When you use DSOPEN, you should know the following things:

- 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-DS9 or a DSCB defined in your program with the DSCB statement. The DSCB must be initialized with a six-character volume name in $DSCBVOL and an eight-character data set name in $DSCBNAM.

Opening a Data Set (DSOPEN) *(continued)*

- To reopen a data set, initialize $DSCBVDE to zero; DSOPEN ignores all other fields.
- If you specify the volume name as six blanks, DSOPEN searches the IPL volume for the data set.
- After DSOPEN completes, #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.
- You can open only one data set on any tape volume at a time.

DSOPEN Example

The following example shows how to open a data set when the data set is not known when the program is loaded. Program MAINPGM, the primary task, prompts the operator for the data set name and volume and calls secondary task OPENPGM. If the operator does not enter volume name, the program assumes the IPL volume.

```
1  MAINPGM PROGRAM  START,MAIN=YES
2          EXTRN    OPENPGM
3  START   MOVEA    #1,DS1
4  READDS  READTEXT RESPONSE,'@@ENTER DSNAME,VOLUME - '
5          IF       (RESPONSE-1,EQ,X'00',BYTE),THEN
             GOTO READDS
           ENDIF
6          MOVE     ($DSCBVOL,#1),IPLVOL,(6,BYTE)
7          MOVE     WHERE,0
8          FIND     C',',RESPONSE,15,WHERE,DSONLY
9          MOVE     #2,WHERE
10         MOVE     ($DSCBVOL,#1),(1,#2),(6,BYTE)
11         MOVE     (0,#2),BLANK8,(8,BYTE)
12 DSONLY  MOVE     ($DSCBNAM,#1),RESPONSE,(8,BYTE)
13         CALL     OPENPGM,(DS1)
14         MOVE     CODE,DS1
15         IF       (CODE,NE,-1),THEN
             PRINTEXT  '@ERROR DURING DSOPEN.  RETURN CODE = '
             PRINTNUM  CODE
           ELSE
16           .
             .
           ENDIF
           PROGSTOP
17         COPY     DSCBEQU
18 CODE    DC       F'0'
19 IPLVOL  EQU      *
   BLANK8  DC       CL8'        '
20 WHERE   DC       F'0'
21 RESPONSE TEXT    ' ',LENGTH=15
22         DSCB     DS#=DS1,DSNAME=DUMMY
           ENDPROG
           END
```

1 Begin the program at START and identify this task as the primary task (MAIN=YES).

2 Identify as an external entry the subroutine that this task will call.

3 Place the address of the DSCB in register 1.

4 Prompt the operator for the data set name. When the operator responds, the system places the response in RESPONSE.

5 Test for a null entry. RESPONSE-1 contains the length of the operator's response.

6 Initialize the volume field (DSCBVOL) of the DSCB to blanks.

7 Initialize the comma locator to zero.

Opening a Data Set (DSOPEN) *(continued)*

8 Find a comma in the operator's response. If no comma exists, branch to DSONLY.

9 Move the position of the comma to register 2.

10 Move the volume name to the volume field (DSCBVOL) of the DSCB.

11 Blank the volume name and the comma preceding it.

12 Move the data set name to the data set name field (DSCBNAM) of the DSCB.

13 Call the routine that opens the data set. Pass the address of the DSCB (pointed to by DS1) to the subroutine.

14 Move the return code into CODE.

15 If the return code does not indicate successful completion (-1), print an error message and the return code.

16 Process the data set with READ/WRITE instructions. ($DSCBEND contains the number of records in the data set.)

17 Cause the DSCB equates to be copied into the program.

18 Reserve storage for the subroutine return code.

19 Set up a default value for IPL volume.

20 Reserve storage for an index to be used in locating the comma.

21 Reserve storage for the operator's response.

22 Generate a data set control block (DSCB). Give the data set name field (DSCBNAM) the temporary name DUMMY.

Program OPENPGM consists of a subroutine and error exit routines for DSOPEN. The subroutine calls DSOPEN.

```
 1  OPENPGM  PROGRAM  MAIN=NO
 2           ENTRY    OPENPGM
 3           SUBROUT  OPENPGM,ADSN
 4           MOVE     SAVE1,#1
 5           MOVE     SAVE2,#2
 6           MOVE     #1,ADSN
 7           MOVE     (0,#1),-1
 8           MOVEA    $DSNFND,LIBEXIT
 9           MOVEA    $DSBVOL,VOLEXIT
10           MOVEA    $DSIOERR,IOEXIT
11           CALL     DSOPEN,ADSN
             GOTO     RETURN
12  LIBEXIT  EQU      *
13           MOVE     #1,ADSN
14           MOVE     (0,#1),1
             PRINTEXT '@DATA SET NOT FOUND DURING DSOPEN@'
             GOTO     RETURN
15  VOLEXIT  EQU      *
16           MOVE     #1,ADSN
17           MOVE     (0,#1),2
             PRINTEXT '@VOLUME NOT FOUND DURING DSOPEN@'
             GOTO     RETURN
18  IOEXIT   EQU      *
19           MOVE     #1,ADSN
20           MOVE     (0,#1),3
             PRINTEXT '@ERROR ENCOUNTERED DURING DSOPEN@'
             GOTO     RETURN
21  RETURN   MOVE     #1,SAVE1
22           MOVE     #2,SAVE2
23           RETURN
24           COPY     TCBEQU
25           COPY     PROGEQU
26           COPY     DDBEQU
27           COPY     DSCBEQU
28           COPY     DSOPEN
29  DISKBUFR DC       128F'0'
30  SAVE1    DC       F'0'
31  SAVE2    DC       F'0'
             END
```

1 Identify the name of the subroutine as OPENPGM. Specify that it is not the main program (MAIN=NO).

Opening a Data Set (DSOPEN) *(continued)*

2 Identify the name of the subroutine as an entry. (In conjunction with the EXTRN statement in the main program, this statement allows the linkage editor to resolve external references.)

3 Define a subroutine with the name OPENPGM. Define a parameter (ADSN) that is passed by the calling program.

4 Save index register 1.

5 Save index register 2.

6 Move the parameter that was passed from the calling program (the address of the DSCB) to register 1.

7 Initialize the return code to indicate successful completion (-1).

8 Move the address of the data-set-not-found routine to the proper error exit within DSOPEN.

9 Move the address of the invalid-volume routine to the proper error exit within DSOPEN.

10 Move the address of the I/O error routine to the proper error exit within DSOPEN.

11 Call DSOPEN, passing the address of the DSCB.

12 Indicate the beginning of the data-set-not-found exit routine.

13 Move the address of the DSCB to register 1.

14 Move a 1 to the first word of the DSCB, indicating data set not found.

15 Indicate the beginning of the invalid-volume exit routine.

16 Move the address of the DSCB to register 1.

17 Move a 2 to the first word of the DSCB, indicating an invalid volume.

18 Indicate the beginning of the I/O error exit routine.

19 Move the address of the DSCB to register 1.

20 Move a 3 to the first word of the DSCB, indicating an I/O error.

21 Restore index register 1.

22 Restore index register 2.

23 Return to the calling program.

24 Cause the TCB equates to be copied into the program.

25 Cause the PROGRAM equates to be copied into the program.

26 Cause the DDB equates to be copied into the program.

27 Cause the DSCB equates to be copied into the program.

28 Cause the DSOPEN equates to be copied into the program.

29 Reserve a 256-byte area for DSOPEN. (This area must have the label DISKBUFR.)

30 Reserve an area in which to save register 1.

31 Reserve an area in which to save register 2.

Coding for Volume Independence

You may code your applications so that they are independent of the volume in which they reside. To achieve volume independence, place all programs and data sets in a single volume on any system and specify the characters ## in the volume name field of any DS= operand or PGMS= operand of the PROGRAM statement. (For information on the PROGRAM statement, refer to the *Language Reference*.)

You can also insert the volume name from which your program was loaded into any DSCB you have coded in your program. If you insert the volume name into a DSCB, you must do so before invoking DSOPEN or $DISKUT3. The volume name, a six-byte field, is located in the $PRGVOL field of the program header.

Opening a Data Set (DSOPEN) *(continued)*

The following example shows a routine that retrieves the volume name and invokes DSOPEN to open the data set JOURNAL, located in the same volume from which the program was loaded.

```
            COPY   TCBEQU
            COPY   PROGEQU
            COPY   DDBEQU
            COPY   DSCBEQU
            COPY   DSOPEN
            .
            .
1  ENTER    TCBGET TCBADDR
2           MOVE   #1,TCBADDR
3           MOVE   #2,($TCBPLP,#1)
4           MOVEA  #1,INDS
5           MOVE   ($DSCBVOL,#1),($PRGVOL,#2),(6,BYTE)
6           CALL   DSOPEN,(INDS)
            .
            .
7           DSCB   DS#=INDS,DSNAME=JOURNAL
8  DISKBUFR DC     128F'0'
9  TCBADDR  DC     F'0'
```

1 Get the address of the task control block (TCB).

2 Move the address of the TCB into register 1.

3 Move the address of the program header into register 2.

4 Move the address of the data set control block (DSCB) into register 1.

5 Move the volume into the DSCB.

6 Call DSOPEN, passing the DSCB as a parameter.

7 Define the DSCB.

8 Define a work area for DSOPEN.

9 Define an area for the TCB address.

Setting Logical End of File (SETEOD)

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 will occur at either the physical or logical end that was set by some previous use of the data set.

The relative record number of the last full physical record is placed 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 $DSCBEDB 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.

SETEOD can be used 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.

SETEOD requires that the DSOPEN copy code, PROGEQU, TCBEQU, DDBEQU, and DSCBEQU be copied in your program.

Setting Logical End of File (SETEOD) *(continued)*

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 as a subroutine through the Event Driven Language CALL statement, passing the DSCB and an I/O error exit routine pointer as parameters.

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

where:

DS1 Names a previously opened DSCB

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

Finding the Device Type (EXTRACT)

The ***inline*** copy code routine EXTRACT determines the device type from the device descriptor block. This routine is provided for applications that are sensitive to device type. For example, an application may need to allocate a data set unless the data set were to reside on a tape. Before attempting to execute instructions that would not execute successfully, the EXTRACT routine may be used to determine the device type.

To use EXTRACT, you must copy the source code inline into your program. The routine requires the address of a DSCB in #1 and returns the device type in #1.

```
MOVEA #1,DS1
COPY  EXTRACT
IF    (#1,EQ,X'3186'),GOTO,TAPEDS
```

In this example, X'3186' is the device ID of an IBM 4969 Magnetic Tape.

To get a list of the device IDs on your system, use the LD command of the $IOTEST utility.

Notes

Chapter 11. Reading and Writing to Tape

This chapter describes the tape facilities you can use when using tape as part of your EDL program.

For information on how to allocate tape data sets, copy data sets from one medium to another, and change tape attributes, refer to the $TAPEUT1 utility in the *Operator Commands and Utilities Reference* or the *Operation Guide*.

For more information on how to access magnetic tape data sets, refer to the *Language Reference*.

For information on data set naming conventions, refer to the "Specifying Data Sets" on page PG-97.

What Is a Standard-Label Tape?

A standard-label tape consists of data sets separated by 80-character label records and tapemarks.

A *label record* is a record that the system writes on a tape to do such things as identify the volume, indicate the beginning of a data set, and indicate the end of a data set.

Standard label tapes contain a volume label (VOL1) and a header label (HDR1) before each data set and a trailer label (EOF1) after each data set. For the contents of the labels, see Appendix A, "Tape Labels" on page PG-263.

A *tapemark* is a control character that the system writes on a tape. The hardware uses tapemarks to recognize such things as the beginning or end of a data set.

What Is a Standard-Label Tape? *(continued)*

You would use standard-label tapes to maintain data security or to control an extensive library of tapes.

What Is a Nonlabeled Tape?

A nonlabeled tape consists of data sets separated only by tapemarks.

Nonlabeled tapes allow you to read tapes that have unknown record length or an unknown label.

You would use nonlabeled tapes if you do not need to maintain strict data security or if you use only a small number of tapes.

Processing Standard-Label Tapes

This section describes how to:

- Read a standard-label tape
- Write a standard-label tape
- Close a standard-label tape
- Bypass standard labels
- Process a tape containing more than one data set

Reading a Standard-Label Tape

The READ instructions allows you to retrieve a record from 18 to 32767 bytes long.

In the following example:

```
TASK04   PROGRAM  START,DS=(UPDATES,(MASTER,56390))
         .
         .
         READ     DS2,BUFF,1,120,END=NMRCDS,ERROR=OOPS,WAIT=YES
         .
         .
BUFF     DATA     60F'0'
```

the system reads one record (indicated by 1 in the third operand) from the second file listed on the PROGRAM statement (data set MASTER on volume serial 56390) into BUFF. (The term *volume serial* means the same as the term *volume*.)

The size of the record is 120 bytes (indicated by 120 in the fourth operand). If no more records exist on the data set, control transfers to NMRCDS. If an error occurs, control transfers to OOPS. The system waits (WAIT=YES) for the read operation to complete before executing the next sequential instruction.

The following READ instruction reads 2 records into BUFF2. BUFF2 must be 654 bytes long.

```
TASK37   PROGRAM BEGIN,DS=((UPDATES,73499),(MASTER,56390))
         .
         .
         READ    DS1,BUFF2,2,327,END=END1,ERROR=ERR,WAIT=YES
         .
         .
BUFF2    DATA    327F'0'
```

The system reads two records (indicated by 2 in the third operand) from the first data set (UPDATES on volume serial 73499) listed on the PROGRAM statement. The size of the record is 327 bytes (indicated by 327 in the fourth operand). If no more records exists on the data set, control transfers to END1. If an error occurs, control transfers to ERR. The system waits (WAIT=YES) for the read operation to complete before executing the next sequential instruction.

Writing a Standard-Label Tape

The WRITE instruction allows you to write a record from 18 to 32767 bytes long.

In the following example:

```
TASK04   PROGRAM START,DS=(UPDATES,(MASTOUT,00032))
         .
         .
         WRITE   DS2,BUFF,1,120,ERROR=GOOF,WAIT=YES
         .
         .
BUFF     DATA    60F'0'
```

the system writes one record (indicated by 1 in the third operand) to the second file listed on the PROGRAM statement (data set MASTOUT on volume serial 00032) from BUFF. The size of the record is 120 bytes (indicated by 120 in the fourth operand). If an error occurs, control transfers to GOOF. The system waits (WAIT=YES) for the write operation to complete before executing the next sequential instruction.

The following WRITE instruction writes 2 records from BUFF2. BUFF2 must be 654 bytes long.

```
TASK74   PROGRAM BEGIN,DS=((DATES,28345),(MASTER,56390))
         .
         .
         WRITE   DS1,BUFF2,2,327,ERROR=ERROR,WAIT=YES
         .
         .
BUFF2    DATA    327F'0'
```

Processing Standard-Label Tapes *(continued)*

The system writes two records (indicated by 2 in the third operand) to the first data set (DATES on volume serial 28345) listed on the PROGRAM statement. The size of the record is 327 bytes (indicated by 327 in the fourth operand). If an error occurs, control transfers to ERROR. The system waits (WAIT=YES) for the read operation to complete before executing the next sequential instruction.

Closing Standard-Label Tapes

Whether you read or write a standard-label tape, you should close the tape data set when you finish reading or writing. Closing a tape data set causes the system to write trailer labels. Use the CONTROL instruction to close a tape data set as follows:

```
TASK98   PROGRAM BEGIN,DS=((DATES,28345),(MASTER,56390))
         .
         .
         CONTROL DS1,CLSOFF
         .
         .
```

The system closes the first data set (DATES on volume serial 28345) listed on the PROGRAM statement. CLSOFF causes the system to rewind the tape and set the tape drive offline.

For information on other ways to close a tape, refer to *Language Reference*.

Bypassing Labels

If you want to bypass the labels on a standard-label tape, you must have defined a tape drive as BLP during system generation or changed the label processing attribute with the $TAPEUT1 utility. For information on defining a BLP drive, refer to *Installation and System Generation Guide*.

The following sample program shows how to bypass standard labels.

```
1 PROG8    PROGRAM  START,DS=((XYZ,TAPE01))
  START    EQU      *
2          READ     DS1,BUFFER,1,80,ERROR=ERR1
3          READ DS1,BUFFER,1,80,ERROR=ERR1
4          CONTROL DS1,FSF
  LOOP     EQU      *
5          READ DS1,BUFFER,1,50,ERROR=ERR2,END=ALLDONE
           GOTO     LOOP
  ALLDONE  EQU      *
6          READ DS1,BUFFER,1,80,ERROR=ERR1
  ENDIT    EQU      *
           PROGSTOP
  ERR1     EQU      *
           PRINTEXT '@LABEL ERROR - RC= '
           PRINTNUM DS1
           GOTO     ENDIT
  ERR2     EQU      *
           PRINTEXT '@READ ERROR - RC= '
           PRINTNUM DS1
           QUESTION '@DO YOU WANT TO CONTINUE? ',                       C
              YES=LOOP,NO=ENDIT
  BUFFER   DATA     40F'0'
           ENDPROG
           END
```

1 Identify the tape as data set XYZ on tape ID TAPE01. The system ignores the data set name but you must supply it.

2 Read the first of the standard label records (the VOL1 label) into BUFFER. (You can insert instructions after this instruction to process the label.)

3 Read the second of the standard label records (the HDR1 label) into BUFFER. (You can insert instructions after this instruction to process the label.)

4 Forward space the file one tapemark. This instruction causes the system to skip any remaining blocks in the header and position itself at the first record of the file.

5 Process the data. This instruction reads a 50-character record (indicated by 50 in the third operand) into BUFFER. If an error occurs, control transfers to ERR2. If no more records exist on the data set, control transfers to ALLDONE.

6 Read the trailer label (the EOF1 label) into BUFFER. You can insert instructions after this instruction to process the label.

Processing Standard-Label Tapes *(continued)*

Processing a Tape Containing More than One Data Set

To process a tape that contains more than one data set, use the $VARYON operator command to position the tape to the data set you want to read. For example, to position a tape at address 4C to the fourth data set, issue the following command:

```
$VARYON 4C 4
```

The system responds as follows:

```
TAPE01 ONLINE
```

TAPE01 is the ID that was assigned to the tape drive at system generation.

After you use the $VARYON operator command, you can process the data set as you would any other tape data set.

Reading a Multivolume Data Set

To read a multivolume data set, you must add instructions to your program to process the data set. The following program reads a multivolume data set.

```
1   PROGX    PROGRAM START,DS=??
    START    EQU      *
2            READ     DS1,BUFFER,1,80,ERROR=ERR1,END=CHKEND
             .
             .
             GOTO       START
    ENDIT    EQU        *
             PROGSTOP
    CHKEND   EQU        *
3            CONTROL DS1,CLSOFF
4            IF     (DS1,EQ,33)
5            PRINTEXT '@EOV ENCOUNTERED - ENTER VOL1 OF NEXT VOLUME@'
6            READTEXT NEWVOL
7            MOVEA #1,DS1
8            MOVE   ($DSCBVOL,#1),NEWVOL,(3,WORD)
9            MOVEA $DSNFND,ERRDSN
             MOVEA     $DSBVOL,ERRVOL
             MOVEA     $DSIOERR,ERRIO
10           QUESTION '@REPLY Y WHEN THE NEXT VOLUME IS MOUNTED AND ONLINE@',
                   NO=ENDIT
11           CALL  DSOPEN,(DS1)
12           GOTO      START
             ENDIF
             GOTO      ENDIT
   ERRDSN    EQU       *
             MOVEA     MSGX,MSG1
             GOTO      ERRMSG
   ERRVOL    EQU       *
             MOVEA     MSGX,MSG2
             GOTO      ERRMSG
   ERRIO     EQU       *
             MOVEA     MSGX,MSG3
   ERRMSG    EQU       *
             PRINTEXT '@DSOPEN ERROR -@'
             PRINTEXT MSG1,P1=MSGX
             PRINTEXT SKIP=1
             GOTO      ENDIT
   MSG1      TEXT      'DATA SET NOT FOUND'
   MSG2      TEXT      'VOLUME NOT FOUND'
   MSG3      TEXT      'I/O ERROR'
   ERR1      EQU       *
             PRINTEXT '@READ ERROR - RC='
             PRINTNUM DS1
             GOTO      ENDIT
```

Processing Standard-Label Tapes *(continued)*

```
BUFFER   DATA     40F'0'         80 BYTE BUFFER
NEWVOL   TEXT     '      '       HOLDS NEW VOLUME #
REPLY    TEXT     LENGTH=2
         COPY     DSOPEN
         COPY     DSCBEQU
         COPY     PROGEQU
         COPY     DDBEQU
DISKBUFR DC       128F'0'
         ENDPROG
         END
```

1 Cause the system to issue a prompt for the data set name and volume of the input data set.

2 Read an 80-character record into BUFFER. If an error occurs transfer control to ERR1. If no more records exist, transfer control to CHKEND.

3 Close the input data set, rewind the tape, and set the tape drive offline.

4 Test for a return code of 33, indicating that the system found an end-of-volume label.

5 Prompt the operator for the volume serial of the next tape.

6 Read the volume serial into NEWVOL.

7 Move the address of the DSCB for the data set into software register 1.

8 Move the volume serial into the $DSCBVOL field of the DSCB.

9 Set the DSOPEN error exits in this instruction and in the next two instructions.

10 Prompt the operator for a response when he/she has mounted the tape.

11 Call the DSOPEN routine to open the next volume of the data set.

12 Resume processing the data.

Processing Nonlabeled Tapes

This section describes how to:

- Define a nonlabeled tape
- Initialize a nonlabeled tape

- Read a nonlabeled tape
- Write a nonlabeled tape

Defining a Nonlabeled Tape

To read and write from a nonlabeled tape, you must define the drive as nonlabeled. If the tape drive hasn't already been defined as nonlabeled, you must:

1. Vary the tape drive offline.
2. Change the label processing attribute to nonlabeled using the $TAPEUT1 utility.
3. Vary the tape drive online.

To vary the tape drive offline, use the $VARYOFF operator command as follows:

```
$VARYOFF 4C
TAPE01 OFFLINE
```

The command varies offline the tape drive at address 4C. TAPE01 is the ID that was assigned during system generation.

The following example shows how to use the $TAPEUT1 utility to change the label processing attribute:

```
$L $TAPEUT1

COMMAND (?)  CT

ENTER TAPEID (1-6 CHARS): TAPE01
TAPE TAPE01 AT ADDR 4C is SL 1600 BPI

DO YOU WISH TO MODIFY?: Y

LABEL (NULL,SL,NL,BLP)?: NL
DENSITY (NULL,800,1600)?: 800

TAPE TAPE01 AT ADDRESS 4C IS NL 800 BPI

COMMAND ?  EN
```

This example changes tape TAPE01 to nonlabeled 800 bytes per inch.

To vary the tape drive online, use the $VARYON operator command as follows:

Processing Nonlabeled Tapes *(continued)*

```
$VARYON 48
TAPE01 ONLINE
```

The command varies online the tape drive at address 48. TAPE01 is the ID that was assigned during system generation.

Initializing a Nonlabeled Tape

To initialize a nonlabeled tape, you must:

1. Vary the tape drive offline.
2. Initialize the tape.
3. Vary the tape drive online.

To vary the tape drive offline, use the $VARYOFF operator command as follows:

```
$VARYOFF 4C
TAPE01 OFFLINE
```

The command varies offline the tape drive at address 4C. TAPE01 is the ID that was assigned during system generation.

To initialize the tape, use the $TAPEUT1 utility as follows:

```
$L $TAPEUT1

COMMAND (?)  IT

TAPE ADDR (1 - 2 HEX CHARS): 4C

NO LABEL 800 BPI? Y

TAPE INITIALIZED

COMMAND ?  EN
```

Processing Nonlabeled Tapes *(continued)*

To vary the tape drive online, use the $VARYON operator command as follows:

```
$VARYON 4C
TAPE01 ONLINE
```

The command varies online the tape drive at address 4C. TAPE01 is the ID that was assigned during system generation.

Reading a Nonlabeled Tape

The READ instructions allows you to retrieve a record from a nonlabeled tape. The records can be from 18 to 32767 bytes long.

In the following example:

```
TASK04   PROGRAM START,DS=(UPDATES,(MASTER,TAPE01))
         .
         .
         READ    DS2,BUFFER,1,80,END=NOMORE,ERROR=ERROR,WAIT=YES
         .
         .
BUFFER   DATA    60F'0'
```

the system reads one record (indicated by 1 in the third operand) from the second file listed on the PROGRAM statement (data set MASTER on tape ID TAPE01) into BUFFER. The size of the record is 80 bytes (indicated by 80 in the fourth operand). If no more records exist on the data set, control transfers to NOMORE. If an error occurs, control transfers to ERROR. The system waits (WAIT=YES) for the read operation to complete before executing the next sequential instruction.

Writing a Nonlabeled Tape

The WRITE instruction allows you to write a nonlabeled record from 18 to 32767 bytes long.

In the following example:

```
TASK04   PROGRAM START,DS=(UPDATES,(MASTOUT,TAPE01))
         .
         .
         WRITE   DS2,BUFF,1,120,ERROR=GOOF,WAIT=YES
         .
         .
BUFF     DATA    60F'0'
```

the system writes one record (indicated by 1 in the third operand) to the second file listed on the PROGRAM statement (data set MASTOUT on tape ID TAPE01) from BUFF. The size of the record is 120 bytes (indicated by 120 in the fourth operand). If an error occurs, control

Processing Nonlabeled Tapes *(continued)*

transfers to GOOF. The system waits (WAIT=YES) for the write operation to complete before executing the next sequential instruction.

Adding Records to a Tape File (UPDATE)

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 as a subroutine through the CALL instruction, passing the DSCB as a parameter.

```
CALL    UPDTAPE,(DS1)
```

where DS1 is a previously opened DSCB.

After the CALL, you must check the return code in the first word of the DSCB for the tape motion return codes. A -1 return code indicates that the tape is positioned correctly for writing records.

The following example adds 1000 records to a tape data set. The program prompts the operator for the data set name and volume.

```
1 UPDTAP   PROGRAM START,DS=((TAPEDS,??))
   START    EQU     *
2          CALL    UPDTAPE,(DS1)
3          IF      (DS1,NE,-1)
              PRINTEXT  '@ERROR - UPDTAPE RC ='
              PRINTNUM  DS1
              PRINTEXT  SKIP=1
              GOTO      ENDIT
            ENDIF
4          DO      1000,TIMES
5            WRITE  DS1,BUFF,ERROR=ERR
              ADD    BUFFNUM,1
            ENDDO
   ENDIT    EQU     *
            IF      (DS1,EQ,-1)
              PRINTEXT  '@TAPE UPDATED SUCCESSFULLY@'
              CONTROL   DS1,CLSRU
              IF      (DS1,NE,-1)
                PRINTEXT  '@CLOSE ERROR - RC ='
                PRINTNUM  DS1
                PRINTEXT  SKIP=1
              ENDIF
            ENDIF
            PROGSTOP
   ERR      EQU        *
            PRINTEXT   '@WRITE ERROR - RC ='
            PRINTNUM   DS1
            PRINTEXT   SKIP=1
            GOTO       ENDIT
   BUFF     DC         127X'FFFF'
   BUFFNUM  DC         F'1'
            COPY       DSCBEQU
            COPY       TDBEQU
            COPY       DDBEQU
            COPY       UPDTAPE
            ENDPROG
            END
```

1 Cause the system to prompt for the name and volume of the tape data set.

2 Call the subroutine, passing the DSCB as a parameter.

3 Check the return code from the subroutine.

4 Add 1000 records to the tape data set.

5 Write a record to the data set from buffer BUFF. If an error occurs, branch to ERR.

Notes

Chapter 12. Communicating with Another Program (Cross Partition Services)

To communicate with another program, you can use cross partition services. Cross partition services require synchronization logic in your programs but no additional storage in the supervisor.

Communication is possible between two programs within the same partition and between programs in different partitions. Cross partition services permit asynchronous but coordinated execution of application programs running in different partitions.

Use these services when interrelated programs and tasks in your application cannot be accommodated in a single partition.

When your task is attached, its TCB ($TCBADS) is updated to contain the number of the address space in which it is executing. The address space value (the partition number minus one) is also known as the hardware address key. This key, along with an address you supply, is used to calculate the target address used in cross partition services. For some functions, you put the address key of the target partition in $TCBADS.

The following sections contain examples of the different uses of the cross partition services.

Communicating with Another Program (Cross Partition Services)

Loading Other Programs

In the following example, PROGA loads PROGB into partition two and passes the parameters at PROGASW1 to it. When PROGB terminates, the supervisor posts the ECB at ENDWAIT, signaling PROGA that PROGB has ended.

In this example, the system queues the program loaded (PROGB) to the terminal that is enqueued by the loading program (PROGA).

$TCBADS is not modified by the LOAD instruction.

PROGA, the loading program, looks like this:

```
1  PROGA     PROGRAM START,1,MAIN=YES
2  ATLIS     ATTNLIST (CA,PROGASTP)
   PROGASTP EQU      *
3            MOVE     #1,PROGASW1
4            MOVE     (0,#1),1,TKEY=1
             ENDATTN
   START     EQU      *
5            TCBGET   PROGAKEY,$TCBADS
6            LOAD     PROGB,PROGASW1,EVENT=ENDWAIT,LOGMSG=YES,PART=2
7            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
```

Notes on PROGA are as follows:

1 Define the primary task (MAIN=YES). Assign priority 1 to the task.

2 Define an attention-interrupt-handling routine. When the operator enters "CA" and presses the attention key, branch to PROGASTP.

3 Move PROGASW1 into register 1. (When this instruction executes, PROGASW1 contains the address of CANCELSW in PROGB.)

4 Move 1 to address (0,#1). Indicate the address key of the loaded program (TKEY=1). Address (0,#1) points to the address of CANCELSW. In PROGB, the IF instruction finds that CANCELSW contains a 1 and passes control to the label STOP.

5 Put PROGA's address key into PROGAKEY.

6 Load PROGB, passing the parameters beginning at label PROGASW1. Identify the event to be posted when PROGB completes (EVENT=ENDWAIT), indicate that the PROGRAM LOADED message is to appear on the terminal, and load the program into partition 2 (PART=2).

7 If PROGB loads successfully, wait for PROGB to post the event ENDWAIT.

The following program, PROGB, is the program being loaded.

When the operator presses the attention key and enters "CA", the attention-interrupt-handling routine at label CANCEL in PROGA begins executing.

```
1 PROGB    PROGRAM START,509,PARM=2
  START    EQU     *
2          PRINTEXT 'TO CANCEL HIT > CA',SKIP=1
           PRINTEXT SKIP=1
3          MOVEA    PROGAWRK,CANCELSW
4          MOVE     #1,$PARM1
5          MOVE     (0,#1),PROGAWRK,TKEY=$PARM2
6 LOOP     IF       (CANCELSW,EQ,1),GOTO,STOP
             GOTO     LOOP
  STOP     EQU      *
7          PROGSTOP -1,LOGMSG=NO
  PROGAWRK DATA   F'0'
  CANCELSW DATA   F'0'
           ENDPROG
           END
```

1 Specify the length of the parameter list that PROGB receives from PROGA (PARM=2). 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).

2 Display a prompt that tells the operator how to cancel PROGB.

3 Move the address of CANCELSW into PROGAWRK.

4 Move the first parameter (the address of PROGASW1) into software register 1.

5 Move the contents of PROGAWRK to the address (0,#1) in PROGA. The TKEY operand of the MOVE instruction supplies the address key of PROGA.

6 Loop until the operator cancels the program.

7 Post the loading program (PROGA) with a -1. Suppress the PROGRAM ENDED message (LOGMSG=NO).

Note: When you execute a LOAD instruction for an overlay or nonoverlay program, the default terminal address or the currently active terminal address of the program issuing the LOAD is placed in the program header of the loaded program. This address is taken from $PRGCCB in the issuing program's program header and placed into $PRGCCB of the loaded program's program header. This address is a CCB address.

Finding Other Programs

The following example uses the WHERES instruction to find another program and return the address key and the load point of a program.

```
1           WHERES   PROGB,ADDRB,KEY=KEYB
              .
              .
              .
   PROGB     DATA     C'PROGB    '
2 PROGB     DATA     C'PROGB    '
3 ADDRB     DATA     F'0'
4 KEYB      DATA     F'0'
```

1 Find program PROGB. Put the load point address in ADDRB and the address key in KEYB.

2 Define the program to be found (the name you give the program when you link-edit it).

3 Define storage for the load-point address.

4 Define storage for the address key.

Communicating with Another Program (Cross Partition Services)

Finding Other Programs *(continued)*

Starting Other Tasks

You can start a task in another partition with the ATTACH instruction.

In the following example, PROGA starts (or "attaches") the task labeled TASKADDR in PROGB.

```
    PROGA     PROGRAM  START
1             COPY     PROGEQU
2             COPY     TCBEQU
    START     EQU      *
3             WHERES   PROGB,ADDRB,KEY=KEYB
4             IF       (PROGA,EQ,0),THEN
                PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
                GOTO      DONE
              ENDIF
5             TCBGET   #1,$TCBVER
6             MOVE     SAVEKEY,($TCBADS,#1)
7             MOVE     ($TCBADS,#1),KEYB
8             ADD      ADDRB,X'34',RESULT=TASKADDR
9             ATTACH   *,P1=TASKADDR
10            MOVE     ($TCBADS,#1),SAVEKEY
              .
              .
    DONE      PROGSTOP
    SAVEKEY   DATA     F'0'
11  PROGB     DATA     C'PROGB    '
    ADDRB     DATA     F'0'
    KEYB      DATA     F'0'
              ENDPROG
              END
```

1 Copy the PROGRAM equates into the program.

2 Copy the task control block (TCB) equates into the program.

3 Find the load-point address and address key of PROGB. Place the load-point address of PROGB into ADDRB and the address key of the program into KEYB.

4 If the WHERES instruction returns a zero, indicating an error, print an error message and end the program.

5 Place the address of PROGA's task control block (TCB) in software register 1.

6 Save PROGA's address key in SAVEKEY.

7 Move PROGB's address key to the address key field ($TCBADS) of the TCB.

8 Add X'34' to the load point of PROGB. Put the result of the addition in TASKADDR. (PROGA assumes that PROGB defines the task to be attached immediately after the PROGRAM statement. The PROGRAM statement generates 52 bytes (X'34') of code.)

9 Attach the task. Assume that the address of the task to be attached is contained in TASKADDR (calculated by the ADD instruction).

10 Restore PROGA's address key from SAVEKEY.

11 Indicate the name of the program to be found. (The name of the program is the name assigned to it when the program was link-edited.)

The following program contains task NEXT that PROGA attaches. This program must be in storage when PROGA issues the WHERES instruction.

```
  PROGB     PROGRAM  START
1 TASKADDR  TASK     NEXT
2 NEXT      ENQT     $SYSPRTR
            PRINTEXT '@SUBTASK IS ATTACHED'
            .
            .
            DEQT
            ENDTASK
  START     EQU      *
3           PRINTEXT '@PROGB STARTED'
4           WAIT     KEY
            .
            .
            PROGSTOP
            ENDPROG
            END
```

1 Define a task with the name TASKADDR.

2 Enqueue the system printer ($SYSPRTR).

3 Print the message PROGB STARTED.

4 Wait for the operator to press the enter key. (The example assumes that the operator will not press the enter key until the task labeled TASKADDR in PROGB has executed.)

Notes:

1. When an ATTACH instruction is executed, the default terminal address or the currently active terminal address of the task issuing the ATTACH is placed into $TCBCCB.

2. When you issue an ATTACH instruction, the system places into $TCBCCB the default terminal address or the terminal address of the task that issued the ATTACH instruction.

Sharing Resources with the ENQ/DEQ Instructions

You can share serially reusable resources with programs in other partitions by using the ENQ and DEQ instructions.

In the following example, SQROOT is a subroutine that has been link-edited by several other programs. The subroutine is serially reusable because only one program can use the subroutine at a time. PROGA attempts to enqueue the queue control block (QCB) in PROGB. PROGA must enqueue the QCB before it can call the subroutine labeled SQROOT.

```
      PROGA     PROGRAM  START
 1              COPY     TCBEQU
 2              EXTRN    SQROOT
      START     EQU      *
 3              WHERES   PROGB,ADDRB,KEY=KEYB
 4              IF       (PROGA,EQ,0),THEN
                  PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
                  GOTO      DONE
                ENDIF
 5              TCBGET   #1,$TCBVER
 6              MOVE     SAVEKEY,($TCBADS,#1)
 7              MOVE     ($TCBADS,#1),KEYB
 8              ADD      ADDRB,X'34',RESULT=PROGBQCB
 9              ENQ      *,BUSY=CANTHAVE,P1=PROGBQCB
10              CALL     SQROOT
11              DEQ
12              MOVE     ($TCBADS,#1),SAVEKEY
                GOTO     DONE
      CANTHAVE  EQU      *
                PRINTEXT '@RESOURCE BUSY'
                MOVE     ($TCBADS,#1),SAVEKEY
                .
                .
                .
      DONE      PROGSTOP
      SAVEKEY   DATA     F'0'
13    PROGB     DATA     C'PROGB    '
      ADDRB     DATA     F'0'
      KEYB      DATA     F'0'
                ENDPROG
                END
```

1 Copy the task control block (TCB) equates into the program.

2 Identify the subroutine as an external entry (to be resolved at link-edit time).

3 Find the load-point address and address key of PROGB. Place the load-point address of PROGB into ADDRB and the address key of the program into KEYB.

4 If the WHERES instruction returns a zero, indicating an error, print an error message and end the program.

5 Place the address of PROGA's task control block (TCB) in software register 1.

6 Save PROGA's address key in SAVEKEY.

7 Move PROGB's address key to the address key field ($TCBADS) of the TCB.

8 Add X'34' to the load point of PROGB. Put the result of the addition in PROGBQCB. (PROGA assumes that PROGB defines the queue control block (QCB) immediately after the PROGRAM statement. The PROGRAM statement generates 52 bytes (X'34') of code.)

9 Enqueue the subroutine. Assume that the address of the task to be attached is contained in PROGBQCB (calculated by the ADD instruction).

10 Call the SQROOT subroutine.

11 Dequeue the subroutine.

12 Restore PROGA's address key from SAVEKEY.

13 Indicate the name of the program to be found. (The name of the program is the name assigned to it when the program was link-edited.)

The subroutine link-edited with PROGA looks like:

```
SUBROUT   SQROOT
ENTRY     SQROOT
PRINTEXT '@SUBROUTINE HAS BEGUN'
.
.
.
RETURN
END
```

PROGB could look like this:

```
      PROGB      PROGRAM   START
      QCB1       QCB
      START      EQU       *
1                WAIT      KEY
                 PROGSTOP
                 ENDPROG
                 END
```

1 Wait 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 SQROOT subroutine.)

Communicating with Another Program (Cross Partition Services)

Synchronizing Tasks in Other Partitions

You can synchronize two or more tasks in different partitions with the WAIT and POST instructions. The following programs show how to issue a POST instruction to a program in another partition.

The first program, PROGA, finds the second program, PROGB, finds its event control block (ECB), and posts the ECB. In this example, PROGB must be loaded before PROGA.

PROGA assumes that PROGB contains an ECB immediately following the PROGRAM statement.

```
    PROGA     PROGRAM   START
1             COPY      TCBEQU
    START     EQU       *
2             WHERES    PROGB,ADDRB,KEY=KEYB
3             IF        (PROGA,EQ,0),THEN
                PRINTEXT  'PROGRAM NOT FOUND'
                GOTO       DONE
              ENDIF
4             TCBGET    #1,$TCBVER
5             MOVE      SAVEKEY,($TCBADS,#1)
6             MOVE      ($TCBADS,#1),KEYB
7             ADD       ADDRB,X'34',RESULT=PGMBECB
8             POST      *,-1,P1=PGMBECB
9             MOVE      ($TCBADS,#1),SAVEKEY
    DONE      PROGSTOP
10  PROGB     DATA      C'XP12B    '
    SAVEKEY   DATA      F'0'
    ADDRB     DATA      F'0'
    KEYB      DATA      F'0'
              ENDPROG
              END
```

1 Copy the task control block (TCB) equates into the program.

2 Find the program defined at PROGB, put the address of the program in ADDRB, and put the address key of the program in KEYB.

3 If the WHERES instruction returns a zero, print an error message and end the program.

4 Put the address of PROGA's task control block (TCB) in register 1.

5 Save PROGA's address key in SAVEKEY.

6 Move PROGB's address key to the address key field ($TCBADS) of the TCB.

7 Add a hexadecimal 34 to the load point address returned by the WHERES instruction. Put the results of the addition in PGMBECB. (PROGA assumes that PROGB defines

an ECB immediately after the PROGRAM statement. The PROGRAM statement generates 52 bytes (X'34) of code.)

8 Post the ECB with a -1. The operand P1=PGMBECB allows the ECB to be calculated by the ADD instruction.

9 Restore PROGA's address key from SAVEKEY.

10 Indicate the name of the program to be found. The name of the program is the name assigned to it when the program was link-edited.

The following program shows how PROGB receives the POST from PROGA. This program must be in storage when PROGA issues the WHERES instruction.

```
1 PROGB    PROGRAM   START
2 ECB1     ECB
  START    EQU       *
3          WAIT      ECB1
           .
           .
           PROGSTOP
           ENDPROG
           END
```

1 Identify the label at which to start executing (START).

2 Define an event control block (ECB). The program defines the ECB here because it will always be 52 bytes (X'34') from the program load point.

3 Wait for PROGA to post the program.

Moving Data Across Partitions

You can also move data across partitions. The following programs show how to move data to a program in another partition.

The first program, PROGA, finds the second program, PROGB, stores its address key, and moves data to the dynamic storage area of PROGB. In this example, PROGB must be loaded before PROGA.

Moving Data Across Partitions *(continued)*

```
    PROGA     PROGRAM  START
1             COPY     PROGEQU
2             COPY     TCBEQU
    START     EQU      *
3             WHERES   PROGB,ADDRB,KEY=KEYB
4             IF       (PROGA,EQ,0),THEN
                PRINTEXT  'PROGRAM NOT FOUND'
                GOTO      DONE
              ENDIF
5             READTEXT MSG,'@ENTER UP TO 30 CHARACTERS',MODE=LINE
6             MOVE     #2,ADDRB
7             TCBGET   #1,$TCBVER
8             MOVE     PROGBBUF,($PRGSTG,#2),FKEY=KEYB
9             MOVE     SAVEKEY,($TCBADS,#1)
10            MOVE     ($TCBADS,#1),KEYB
11            MOVE     #2,PROGBBUF
12            MOVE     (0,#2),MSG,(30,BYTE),TKEY=KEYB
13            MOVE     ($TCBADS,#1),SAVEKEY
    DONE      PROGSTOP
    MSG       TEXT     LENGTH=30
    PROGBBUF  DATA     F'0'
14  PROGB     DATA     C'PROGB    '
    SAVEKEY   DATA     F'0'
    ADDRB     DATA     F'0'
    KEYB      DATA     F'0'
              ENDPROG
              END
```

1 Copy the PROGRAM equates into the program.

2 Copy the task control block (TCB) equates into the program.

3 Find the program defined at PROGB, put the address of the program in ADDRB, and put the address key of the program in KEYB.

4 If the WHERES instruction returns a zero, print an error message and end the program.

5 Prompt the operator for data and place the operator's response in MSG.

6 Move the address of PROGB in register 2.

7 Put the address of PROGA's task control block (TCB) in register 1.

8 Move the address of PROGB's dynamic storage area to PROGBBUF. Indicate PROGB's address key (FKEY=KEYB). PROGB has STORAGE=256 on its PROGRAM statement. This operand causes the system to acquire a 256-byte area of

storage when it loads PROGB. The address of this area is in PROGB's program header (at $PRGSTG).

9 Save PROGA's address key in SAVEKEY.

10 Move PROGB's address key to the address key field ($TCBADS) of the TCB.

11 Move the address of PROGB's dynamic storage area to register 2.

12 Move the data that the operator entered (MSG) into PROGB's dynamic storage area. Move 30 bytes and indicate the address key of the program to which the data is being moved (TKEY=KEYB).

13 Restore PROGA's address key from SAVEKEY. Note that $TCBADS is immediately restored to its original value. Doing so avoids unpredictable results.

14 Indicate the name of the program to be found. The name of the program is the name assigned to it when the program was link-edited.

The following program shows how PROGB receives the data from PROGA. The program must be in storage when PROGA issues the WHERES instruction.

```
1 PROGB     PROGRAM   START,STORAGE=256
  START     EQU       *
2                     .
                      .
3                     MOVE      #1,$STORAGE
4                     MOVE      MSG2,(0,#1),(30,BYTE)
5                     PRINTEXT  '@THE DATA THAT WAS PASSED WAS'
                      PRINTEXT  MSG2
                      PROGSTOP
  MSG2      TEXT      LENGTH=30
            ENDPROG
            END
```

1 Identify the label at which to start executing (START). Specify 256 bytes of dynamic storage. (Even though the program requires only 30 bytes, the system rounds up to a multiple of 256.)

2 Insert instructions here to wait for PROGA to send data.

3 Move the address of the dynamic storage area (contained in $STORAGE) to register 1.

4 Move 30 bytes from the dynamic storage area to MSG2.

5 Print the data.

$TCBADS is used to calculate the partition and address to/from which data will be transferred.

Communicating with Another Program (Cross Partition Services)

Reading Data across Partitions

You can read data across partitions with the READ instruction.

In the following example, program PROGA reads data and passes it to a buffer in program PROGB. PROGA assumes that PROGB is in another partition.

```
1  PROGA     PROGRAM   START,DS=ACCOUNTS
2            COPY      PROGEQU
3            COPY      TCBEQU
   START     EQU       *
4            WHERES    PROGB,ADDRB,KEY=KEYB
5            IF        (PROGA,EQ,0),THEN
               PRINTEXT  'PROGRAM NOT FOUND',SKIP=1
               GOTO        DONE
             ENDIF
6            MOVE      #2,ADDRB
7            TCBGET    #1,$TCBVER
8            MOVE      PROGBBUF,($PRGSTG,#2),FKEY=KEYB
9            MOVE      SAVEKEY,($TCBADS,#1)
10           MOVE      ($TCBADS,#1),KEYB
11           READ      DS1,*,P2=PROGBBUF
12           MOVE      ($TCBADS,#1),SAVEKEY
   DONE      PROGSTOP
   SAVEKEY   DATA      F'0'
13 PROGB     DATA      C'PROGB    '
   ADDRB     DATA      F'0'
   KEYB      DATA      F'0'
             ENDPROG
             END
```

1 Define data set ACCOUNTS on the IPL volume.

2 Copy the PROGRAM equates into the program.

3 Copy the task control block (TCB) equates into the program.

4 Find the load-point address and address key of PROGB. Place the load-point address of PROGB into ADDRB and the address key of the program into KEYB.

5 If the WHERES instruction returns a zero, indicating an error, print an error message and end the program.

6 Move the address key of PROGB into software register 2.

7 Place the address of PROGA's task control block (TCB) in software register 1.

8 Move 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).

9 Save PROGA's address key in SAVEKEY.

10 Moves PROGB's address key to the address key field ($TCBADS) of the TCB.

11 Read 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.

12 Restore PROGA's address key from SAVEKEY.

13 Indicate the name of the program to be found. (The name of the program is the name you give the program when you link-edit it.)

The following program shows how PROGB receives the data from PROGA. The program must be in storage when PROGA issues the WHERES instruction.

```
1 PROGB     PROGRAM   START,STORAGE=256
  START     EQU       *
                      .
                      .
                      .
2                     MOVE      #1,$STORAGE
3                     MOVE      OUTPUT,(0,#1),(50,BYTE)
4                     PRINTEXT  '@THE DATA RECEIVED FROM PROGA IS :'
5                     PRINTEXT  OUTPUT,SKIP=1
  OUTPUT    TEXT      LENGTH=50
            ENDPROG
            END
```

1 Identify the label at which to start executing (START). Specify 256 bytes of dynamic storage. (Even though the program requires only 50 bytes, the system rounds up to a multiple of 256.)

2 Move the address of the dynamic storage area (contained in $STORAGE) to software register 1.

3 Move 50 bytes of data from the dynamic storage area into OUTPUT.

4 Print a message.

5 Print the data.

Notes

Chapter 13. Communicating with Other Programs (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, 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.

For example, an EDL program, the primary program, loads a system utility such as $COPYUT1. The program cannot distinguish between connection to a real terminal or a virtual terminal. The program uses the READTEXT instruction to read the prompts from the utility. Then it uses the PRINTEXT instruction to send replies to the utility.

Defining Virtual Terminals

To define a virtual terminal connection during system generation, you must:

- Define two TERMINAL configuration statements.
- Include the supervisor module IOSVIRT.

For informationon how to define TERMINAL statements and include IOSVIRT, refer to *Installation and System Generation Guide.*

You can find out if your system has virtual terminals by using the LA command of the $TERMUT1 utility. If your system has virtual terminals, $TERMUT1 lists the virtual terminals as follows:

```
 NAME      ADDR   TYPE       PART   HARDCOPY  ON-LINE

CDRVTA      **     VIRT        1                  YES CONNECTED CDRVTB   SYNC=YES
CDRVTB      **     VIRT        1                  YES CONNECTED CDRVTA
   .
   .
```

The output from $TERMUT1 indicates that CDRVTA is the primary program (SYNC=YES).

The DEVICE and ADDRESS parameters of the TERMINAL statement define the terminals as virtual terminals. The two TERMINAL statements must reference each other, as shown below.

```
CDRVTA    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTB,SYNC=YES
CDRVTB    TERMINAL  DEVICE=VIRT,ADDRESS=CDRVTA
```

The SYNC parameter of terminal CDRVTA designates it as the terminal to which synchronization events will be posted. The synchronization between virtual terminals is discussed in "Interprogram Dialogue" on page PG-217.

Loading from a Virtual Terminal

When an EDX program is loaded from a real terminal, that terminal becomes its "primary" communication port. When one program loads another, the current terminal of the first program is "passed" and becomes the primary terminal of the second. It is this convention that allows a new program to establish a virtual terminal as the primary port for the loaded program. For example:

```
         .
         .
         ENQT      SEC
         LOAD      $TERMUT1,LOGMSG=NO,EVENT=ENDWAIT
         ENQT      PRIM
         .
         .
PRIM     IOCB      CDRVTA
SEC      IOCB      CDRVTB
```

After this sequence, $TERMUT1 has CDRVTB (the "other" end of the channel) as its primary port, and the loading program has CDRVTA ("this" end of the channel) as its current port.

Interprogram Dialogue

Once the connection between the two communicating programs has been established, you can use the PRINTEXT, READTEXT, PRINTNUM and GETVALUE instructions to send and receive data. You can generate attention interrupts with the TERMCTRL instruction. (Refer to the *Language Reference* for information on the TERMCTRL instruction.) The usual conventions with respect to output buffering and advance input apply.

To use virtual terminals, you must know something about communications protocol (such as knowing when a program is ready for input or has ended). You can use the task code word to find out this information.

Sample Program

The following sample program uses 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. (The six bytes in each TEXT statement is preceded by two length/count bytes.)

Each reply is 8 bytes long (six bytes of text plus two length/count bytes). The program issues a READTEXT until $INITDSK prompts for input. Then the program issues a PRINTEXT to send the reply to the $INITDSK prompt. After $INITDSK ends, the program prints a completion message to the terminal.

```
INIT      PROGRAM   BEGIN
          COPY      PROGEQU
A         IOCB      CDRVTA                    SYNC TERMINAL
B         IOCB      CDRVTB
DEND      ECB
BEGIN     EQU       *
          ENQT      B
          LOAD      $INITDSK,LOGMSG=NO,EVENT=DEND
          ENQT      A                    GET SYNC TERMINAL
          MOVEA     #1,REPLIES+2
          DO        7,TIMES              REPLY TO PROMPTS
            DO      UNTIL,(RETCODE,EQ,8)    BREAK CODE
              READTEXT  LINE,MODE=LINE   LOOP FOR PROMPT MSGS
              MOVE      RETCODE,INIT     SAVE RETURN CODE
            ENDDO
           PRINTEXT (0,#1)               SEND REPLY
           ADD      #1,8                 NEXT REPLY
          ENDDO
          READTEXT  LINE,MODE=LINE       PGM END MSG
          WAIT      DEND                 WAIT FOR END EVENT
          DEQT
          PRINTEXT  'EDX003 INITIALIZED'
          PROGSTOP
RETCODE   DATA      F'0'                 RETURN CODE
LINE      TEXT      LENGTH=80
REPLIES   EQU       *
          TEXT      'IV    '             COMMAND?
          TEXT      'EDX003'             VOLUME?
          TEXT      'Y     '             CONTINUE?
          TEXT      '60    '             NBR OF DATA SETS?
          TEXT      'N     '             VERIFY?
          TEXT      'N     '             NUCLEUS?
          TEXT      'EN    '             COMMAND?
          ENDPROG
          END
```

Chapter 14. Designing and Coding Sensor I/O Programs

This chapter provides the information you need to code a sensor I/O application program. Topics covered include:

- Sensor I/O devices
- Symbolic I/O assignments
- Sensor I/O instructions

The chapter also provides several examples.

What is Digital Input/Output?

A unit of digital sensor I/O is a physical group of sixteen contiguous points. The entire group of sixteen points is accessed as a unit on the I/O instruction level: programming support allows logical access down to the single point level.

Digital input (DI) is usually used to acquire information from instruments which present binary encoded output, or to monitor contact/switch status (open/closed). Digital output (DO) is used to control electrically operated devices through closing relay contacts, such as pulsing stepping motors.

Process interrupt (PI) is a special form of digital input. If a point of digital input changes state, and then changes state again, without an intervening READ operation from the program, the

What is Digital Input/Output? *(continued)*

status change will be undetected. With process interrupt, a point changing from the off state to on generates a hardware interrupt, which is then routed through software support to an interrupt-servicing application program that can respond to the external event which caused the interrupt. Process interrupt is often used for monitoring critical or alarm conditions, which must be serviced quickly, the occurrence of which must not go undetected.

What is Analog Input/Output?

A physical unit of analog input (AI) can be a group of eight points or sixteen points, depending on the type. Analog output (AO) is installed in groups of two points. Each point of analog input or analog output is accessed separately.

Analog input is used to monitor devices that produce output voltages proportional to the physical variable or process being measured. Examples include laboratory instruments, strain gauges, temperature sensors, or other nondigitizing instruments. Digital input was described as monitoring an on/off status; only two conditions were possible. With analog input, the information is carried in the amplitude of the voltage sensed rather than in its presence or absence.

The starter supervisor contains no support for sensor I/O. You must do a tailored system generation to include the required support modules in your own supervisor.

Figure 7 on page PG-221 shows how sensor devices are connected to a Series/1 through the 4982 sensor I/O unit. The devices (DI, DO, PI, AO, and AI) attach to a controller, which in turn attaches to the Series/1. The sensor I/O attachment (controller), and each of the devices attaching to it, have unique hardware addresses. In this figure, the physical connections are there, and the hardware addresses are assigned (wired in), but the starter supervisor in storage lacks the support necessary to operate the devices.

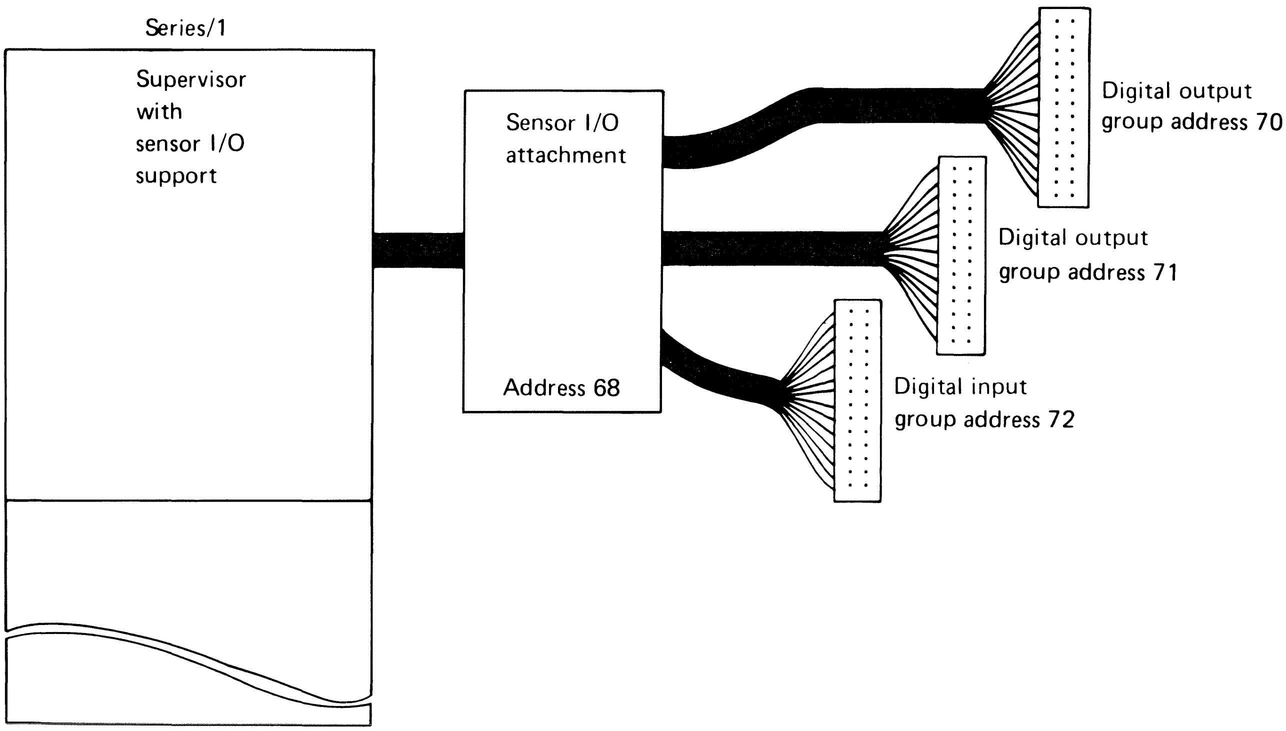


Figure 7. Sensor Device Connections

Building a tailored supervisor involves the assembly of a series of system configuration statements that reflect the I/O configuration you wish to support. For more information on system configuration statements, refer to *Installation and System Generation Guide.* When programs reference these devices, they use symbolic references, rather than actual addresses. The I/O definition statement (IODEF) establishes the logical link between the addresses defined in the supervisor, and the symbols used to read from and write to the devices at those addresses from an application program.

All sensor-based input/output operations are performed by executing a sensor-based I/O (SBIO) instruction. The type of operation is determined by the type of device referenced in the instruction. For more information on the SBIO statement, refer to *Language Reference.* The symbolic reference to a logical device in the SBIO statement is linked to the definition in the IODEF statement, which relates that device to the hardware address specified by the system configuration statement at system generation time.

What are Sensor-Based I/O Assignments?

The sensor-based I/O instruction (SBIO) refers to the I/O devices using a three- or four-character name. The first two characters identify the type of device: AI, DI, PI, AO, and DO for analog input, digital input, process interrupt, analog output, and digital output, respectively. The next one or two characters are the identification for the device, a number between 1 and 99. For example, if you have three analog input terminals, you may identify them as AI1, AI2, and AI3. Before the application program is compiled, the sensor-based I/O definition statement (IODEF) assigns the actual physical addresses. All SBIO instructions are independent of the physical location of the sensor I/O points.

The assignment of sensor I/O symbolic addresses is described under "Providing Addressability (IODEF)" on page PG-223. Figure 8 shows the relationship between sensor-based I/O instructions, definition statements, and configuration statements.

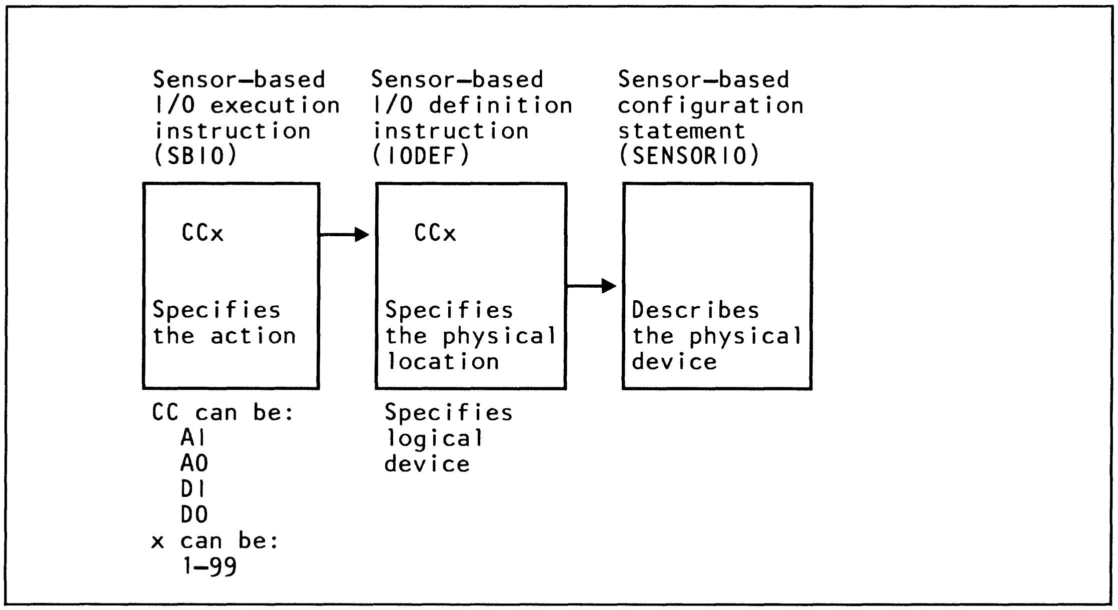


Figure 8. Sensor-Based Symbolic I/O Assignment

Coding Sensor-Based Instructions

This section describes the instructions used in sensor-based I/O applications. The following instructions are defined:

- IODEF - provides addressability by specifying physical location
- SBIO - specifies the I/O operation to be performed

- SPECPIRT - allows control to be returned to the supervisor from a special process-interrupt routine

Providing Addressability (IODEF)

Use the IODEF instruction to provide addressability for the sensor-based I/O facilities which are referenced symbolically in an application program. The specific form used varies with the type of I/O being performed.

Group all IODEF statements of the same form (AI, AO, DI, DO, or PI) together in the program and place them ahead of the SBIO instructions that reference them.

All IODEF statements must be in the same assembly module as the TASK or ENDPROG statement. For high level languages, see the appropriate manual for instructions on how to accomplish this. If the SBIO instructions are to be in a separate module, you can provide addressability using ENTRY/EXTRN statements.

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

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

See the *Language Reference* for the syntax of PI, DO, DI AO, and AI.

Examples

The following IODEF instructions define two process interrupts, a digital output group, a digital output group as external sync, a digital input group, an analog input point, and an analog output point.

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

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

SPECPI - Process Interrupt User Routine

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

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

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

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

Using the Special Process-Interrupt Bit

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

FASTPI1 EQU      *

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

In the following example, control is given to the user at label FASTPI2.

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

FASTPI2   EQU       *
```

Specifying I/O Operations (SBIO)

The SBIO instruction provides communication using analog and digital I/O. Options allow you to:

- Index using a previously defined BUFFER statement.
- Update a buffer address in the SBIO instruction after each operation.
- Use a short form of the instruction, omitting loc (data location) to imply a data address within the SBIOCB.

Options available with digital input and output provide PULSE output and the manipulation of portions of a group with the BITS=(u,v) keyword parameter.

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

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

A sensor based input/output control block (SBIOCB) is inserted into an application program for each referenced sensor I/O device. The SBIOCB, containing a data I/O area and an event control block (ECB), supplies information to the supervisor. When an SBIO instruction executes, the supervisor either stores data (for AI and DI operations) or fetches data (for AO and DO operations) from a location in the IOCB with the label of the referenced I/O point (for example, AI1, DI2, DO33, AO1). An application program can reference these locations the same way any other variable is referenced, allowing you to use the short form of the SBIO instruction (for example, SBIO DI1), and subsequently reference DI1 in other instructions. You may equate a more descriptive label to the symbolic names (for example SWITCH EQU DI15), but the SBIO instruction must use the symbolic name as described above.

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

Coding Sensor-Based Instructions *(continued)*

Reading Analog Input (example)

This example shows SBIO instructions and IODEF statements 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 PTS INTO
                              NEXT 2 LOCATIONS OF 'BUF'
SBIO   AI1,BUF,2              READ THE SAME POINT TWO TIMES
 or                           AND PUT INFORMATION IN TWO
SBIO   AI1,BUF,2,SEQ=NO       LOCATIONS OF BUFF
```

Writing Analog Output (example)

This example shows SBIO instructions and IODEF statements 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
```

Reading Digital Input (example)

This example shows SBIO instructions and 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
```

Writing Digital Output (example)

This example shows SBIO instructions and 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
```

Pulse Digital Output (example)

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
```

Returning from the Process-Interrupt Routine (SPECPIRT)

Use the SPECPIRT instruction to return control to the supervisor from a special process interrupt (SPECPI) routine. If the user routine is in partition 1, a branch instruction is used to return. Return from another partition requires execution of a Series/1 assembler SELB instruction after registers R0 and R3 are saved in the level block to be selected. SPECPIRT is used only for TYPE=BIT SPECPI routines. See the description of IODEF (SPECPI) for additional information.

```
label      SPECPIRT

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

Analog Input Sample

This program takes 256 samples from analog input address AI1 at a sampling rate of 10 points/second. Set the run light on in the lab at the start of the run and turn it off at the end. The run light is connected to bit 3 of group DO2.

```
TKNAME    PROGRAM   START
          IODEF     DO2,TYPE=GROUP,ADDRESS=87
          IODEF     AI1,ADDRESS=83
START     SBIO      DO2,1,BITS=(3,1) TURN ON RUN LIGHT
*
          DO        256,TIMES      SET UP FOR 256 PTS
          STIMER    100            SET TIMER FOR 100 MS
          SBIO      AI1,BUFR,INDEX   READ AI1 WITH
*    AUTOMATIC INDEXING INTO THE BUFFER 'BUFR'
*    AND THEN WAIT FOR THE TIMER TO EXPIRE
          WAIT      TIMER
          ENDDO                      END OF LOOP
*
          SBIO      DO2,0,BITS=(3,1) TURN OFF RUN LIGHT
*
*    . . .  CONTINUE PROGRAM
*
BUFR      BUFFER    256              256 WORD BUFFER
```

The program begins by writing a 1 into bit 3 of digital output group DO2. A DO loop initializes for 256 cycles. At this point, a software timer is set up for 100 milliseconds to provide sampling at 10 points/second. The analog data is read into BUFR using the SBIO instruction with automatic indexing. After the data is read, the program waits for the timer to expire before returning for the next sample. When all the data is collected, the run light is turned off by writing a 0 into bit 3 of DO2.

Analog Input With Buffering To Disk

This program takes analog data readings at equal time intervals. The number of data points and the time interval in milliseconds are read in from the operator's terminal. The program will allow from 10 to 10,000 data points to be taken at time intervals between 10 milliseconds and 10 seconds (10,000 msec). The data collection is initiated by a process interrupt start signal. The program is aborted by using the keyboard function 'AB'. Also, a second keyboard function, 'NP', is used to print a status switch. The switch will be equal to zero if the start signal has not been received or equal to the number of data points to be read if the start signal has been received and data collection has begun.

```
*         TITLE 'SAMPLE ANALOG DATA ACQUISITION PROGRAM'
*
*
READATA   PROGRAM BEGIN,DS=??
          ATTNLIST (AB,ABORT,NP,SWPRNT)
*
*     ABORT THE EXPERIMENT
*
ABORT     MOVE  SWITCH,1
          ENDATTN
*
*     PRINT OUT EXPERIMENT SWITCH
*
SWPRNT    PRINTEXT TXT10
          PRINTNUM SWITCH
          PRINTEXT SKIP=1
          ENDATTN
*
          IODEF    AI1,ADDRESS=91,POINT=0
          IODEF    PI1,ADDRESS=94,BIT=15
*
*     EXPERIMENT INITIALIZATION
*
BEGIN     PRINTEXT TXT1
          GETVALUE RUNUM,TXT2 REQUEST RUN IDENTIFIER
GETINT    GETVALUE INTVL,TXT3  REQUEST TIME INTERVAL
          IF  (INTVL,LT,10),OR,(INTVL,GT,10000),GOTO,GETINT
GETPTS    GETVALUE NPTS,TXT4   REQUEST NO. OF POINTS
          IF  (NPTS,LT,10),OR,(NPTS,GT,10000),GOTO,GETPTS
*
          WRITE DS1,RUNUM      RUN PARAMETERS IN 1ST SECTOR
          RESET SWITCH
```

Coding Sensor-Based Instructions *(continued)*

```
         PRINTEXT TXT9        PRINT READY MESSAGE
         WAIT   PI1,RESET      WAIT FOR START SIGNAL
         MOVE   SWITCH,NPTS    SET SWITCH TO NPTS
*    THIS IS THE DATA ACQUISITION PORTION OF THE PROGRAM
*
         DO     NPTS           LOOP COUNT SET ABOVE
         STIMER INTVL      TIME INTERVAL SET ABOVE
         SBIO   AI1,BUFFER,INDEX  READ A DATA POINT
         IF     (BUFINDEX,EQ,128),GOTO,ATTACH  1ST BUFFER
                                                    FULL?
         IF     (BUFINDEX,NE,256),GOTO,TWAIT   NO, IS 2ND
                                                    FULL?
         MOVE   BUFINDEX,0      ..YES, RESET BUFFER INDEX
         ADD    POINTCNT,256    INCREMENT DATA COUNTER
*
ATTACH   IF     (DISK,NE,-1),GOTO,STOP    IS DISK TASK
                                              ATTACHED?
*   START DISK OUTPUT TASK
         ATTACH DISKTASK
*
TWAIT    WAIT   TIMER       WAIT FOR END OF TIME INTERVAL
         IF     (SWITCH,EQ,1),GOTO,STOP  TEST FOR 'ABORT'
ENDLOOP  ENDDO
*
         IF  (BUFINDEX,EQ,0),OR,(BUFINDEX,EQ,128),GOTO,STOP
         WAIT   DS1          ..YES, WAIT FOR DISK WRITE
         ADD    POINTCNT,BUFINDEX  UPDATE DATA COUNTER
         ATTACH DISKTASK       START LAST DISK OUTPUT
*
STOP     WAIT   DS1        WAIT FOR LAST OUTPUT OPERATION
         ENQT                  GET CONTROL OF TERMINAL
         PRINTEXT TXT6         PRINT TERMINATING MESSAGE
         PRINTNUM POINTCNT
         PRINTEXT TXT7
         DEQT                  RELEASE TERMINAL
         PROGSTOP
```

```
*     THIS IS THE DATA RECORDING TASK.  IT IS ATTACHED BY
*     THE DATA ACQUISITION TASK EACH TIME THAT 128 WORDS OF
*     DATA HAVE BEEN READ IN.  ONE PORTION OF THE BUFFER WILL
*     BE TRANSFERRED TO DISK WHILE DATA IS SIMULTANEOUSLY
*     BEING READ INTO THE OTHER PORTION OF THE BUFFER.

*     THIS TASK RUNS ON LEVEL 3 AT A LOWER PRIORITY THAN
*     THE DATA ACQUISITION TASK IN ORDER TO MAXIMIZE
*     TIMING ACCURACY.
DISKTASK TASK   DISK1,300,EVENT=DISK
DISK1    WRITE DS1,BUFFER1,ERROR=DISKERR
         DETACH -1             ..OK
         WRITE DS1,BUFFER2,ERROR=DISKERR
         DETACH -1             ..OK
         GOTO   DISK1
*               PRINT DISK ERROR MESSAGE
*
DISKERR  MOVE   ERROR,DISKTASK SAVE ERROR CODE
         ENQT                  GET CONTROL OF TERMINAL
         PRINTEXT TXT5
         PRINTNUM ERROR
         PRINTEXT  SKIP=1
         DEQT                  RELEASE TERMINAL
         ENDTASK 1              DETACH WITH CODE = 1
*
*
*               DATA AND CONSTANTS
*
TXT1     TEXT '@SAMPLE ANALOG DATA ACQUISITION PROGRAM@'
TXT2     TEXT '@ENTER RUN NUMBER '
TXT3     TEXT '@ENTER INTERVAL IN MS (10-10000) '
TXT4     TEXT '@ENTER NO. OF POINTS (10-10000) '
TXT5     TEXT '@DISK ERROR '
TXT6     TEXT '@RUN ENDED AFTER '
TXT7     TEXT ' POINTS@'
TXT9     TEXT '@READY FOR PI SIGNAL TO BEGIN TAKING DATA@'
TXT10    TEXT '@EXPERIMENT SWITCH = '

POINTCNT DATA  F'0'               NUMBER OF POINTS TAKEN
SWITCH   DATA  F'0'               SET TO '1' FOR 'ABORT'
RUNUM    DATA  F'0'               RUN IDENTIFIER
INTVL    DATA  F'0'               TIME INTERVAL
NPTS     DATA  F'0'               NUMBER OF POINTS TO TAKE
ERROR    DATA  F'0'
BUFFER   BUFFER 256,INDEX=BUFINDEX   DATA BUFFERS
BUFFER1  EQU    BUFFER         FIRST 128 WORDS
BUFFER2  EQU    BUFFER+256     SECOND 128 WORDS
*
         ENDPROG
         END
```

Digital Input and Averaging

This example illustrates the programming of a simple time averaging application. The program reads digital input group DI1 every time a process interrupt occurs on PI2. One complete scan is 128 data points. Each scan is added to a double-precision averaging buffer. The number of scans is read from the terminal as an initialization parameter. Also, the program asks whether to reset the averaging buffer before starting to scan. The maximum number of scans must be less than 1000.

```
START     GETVALUE  NSCAN,TXT1     GET NO. OF SCANS
          IF        (NSCAN,GE,1000),GOTO,ERROR
          RESET     PI2
          QUESTION  TXT2,NO=BEGIN  RESET AVERG. BUFFER?
          MOVE      ABUFR,0,256    YES - RESET IT
BEGIN     DO        NSCAN          SET UP FOR NSCANS
          DO        128            SET FOR 128 POINTS
          WAIT      PI2            WAIT FOR INTERRUPT
          RESET     PI2            RESET INTERRUPT
          SBIO      DI1,BUFR,INDEX   READ DI1(INDEXING)
          ENDDO
*
*        ONE SCAN COMPLETE - MOVE DATA TO AVERG BUFFER
*
          ADDV      ABUFR,BUFR,128,PREC=D
          MOVE      I,0           RESET BUFFER INDEX
          ENDDO
*
*         ALL SCANS COMPLETE
          PRINTEXT  TXT3
              .
              .
              .      THE REST OF THE PROGRAM

TXT1      TEXT      '@NUMBER OF SCANS - '
TXT2      TEXT      ' RESET AVERAGING BUFFER? '
TXT3      TEXT      ' ALL SCANS COMPLETE@'
NSCAN     DATA      F'0'
BUFR      BUFFER    128,INDEX=I
ABUFR     BUFFER    256
*
ERROR     PRINTEXT  TXT4      PRINT ERROR MESSAGE
          GOTO      START     RETURN FOR INPUT
TXT4      TEXT      ' TOO MANY SCANS - RE-ENTER@'
```

In this example, the number of scans to be done is read from the terminal and checked against 1000. If it is greater than or equal, an error message is printed and the program returns for a new input parameter. The operator is asked if the averaging buffer is to be reset. If yes, the MOVE instruction sets the averaging buffer (ABUFR) to 0. A loop is then initialized for the number of scans desired. A second loop is set up for a single scan of 128 points. The program waits for an interrupt on PI2 and, when it occurs, resets the interrupt for the next point, reads the digital input DI1 using automatic indexing into the buffer BUFR. When a scan is complete, the data is added to the ABUFR buffer. The buffer index, I, is reset to 0. When all scans are complete, a message is printed. The output from the program is illustrated in the following example:

```
NUMBER OF SCANS - 33
 RESET AVERAGING BUFFER? Y
 ALL SCANS COMPLETE
```

Chapter 15. Designing and Coding Graphic Programs

The Event Driven Executive provides various graphics-oriented tools that can assist you in the development of a graphics application.

The graphics tools you can use are the EDL graphics instructions and the graphics utilities. This section describes the graphic instructions supported by the Event Driven Executive. The graphic utilities are described in the *Operator Commands and Utilities Reference*.

Graphics Instructions

Seven graphics instructions are provided by the Event Driven Executive. These graphics instructions, used with the terminal support described, can aid in the preparation of graphic messages, allow interactive input, and draw curves on a display terminal.

These instructions are only valid for ASCII terminals that have a point-to-point vector graphics capability and are compatible with the coordinate conversion algorithm described in *Internal Design* for graphics mode control characters. The function of the various ASCII control characters used by a terminal are described in the appropriate device manual. Such terminals may be connected to the Series/1 via the #7850 Teletypewriter Adapter.

Use the graphics instructions in the same manner as other Event Driven Language instructions, except that the supporting code is included in your program rather than in the supervisor. If you code all the instructions in a program, this code requires approximately 1500 bytes of storage.

Graphics Instructions *(continued)*

When using the graphics instructions described, detailed manipulation of terminal instructions and text messages is not required.

All graphics instructions deal with ASCII data. Therefore, when you send an ASCII text string to the terminal, code the XLATE=NO parameter on the PRINTEXT instruction.

Use of the graphics instructions requires that your object program be processed by the linkage editor, $EDXLINK, to include the graphics functions which are supplied as object modules. Refer to Chapter 5, "Preparing an Object Module for Execution" on page PG-81 for the description of the autocall option of $EDXLINK, and for information on the use of the "AUTO=$AUTO,ASMLIB" option of $EDXLINK.

The following is a list of the graphics instructions provided by the Event Driven Executive. These instructions are described in detail in the *Language Reference*.

- The CONCAT statement concatenates two text strings or a text string and a graphic control character.
- The GIN instruction allows you to specify unscaled coordinates interactively, rings the bell, displays cross hairs, waits for the operator to position the cross hairs and key in any single character, returns the coordinates of the cross-hair cursor, and optionally returns the character entered by the user.
- The PLOTGIN instruction allows you to specify scaled coordinates, rings the bell, displays the cross hairs, and waits for the operator to position the cross-hairs and key any character.
- The SCREEN instruction converts x and y numbers representing a point on the screen of a terminal to the 4-character text string which will be interpreted by the terminal as the graphic address of the point.
- The XYPLOT instruction is used to draw a curve on the display connecting points specified by arrays of x and y values.
- The YTPLOT instruction draws a curve on the display connecting points equally spaced horizontally and having heights specified by an array of y values. Data values are scaled to screen addresses according to the plot control block, and points outside the range are placed on the boundary of the plot area.

The Plot Control Block

The plot control block is required by the PLOTGIN, XYPLOT, and YTPLOT instructions.

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

All 8 explicit values (no addresses) are required and have the following meaning:

xls x screen location at left edge of plot area

xrs x screen location at right edge of plot area

xlv x data value plotted at left edge of plot

xrv x data value plotted at right edge of plot

ybs y screen location at bottom edge of plot

yts y screen location at top edge of plot

ybv y data value plotted at bottom edge of plot

ytv y data value plotted at top edge of plot

The Plot Control Block *(continued)*

Example

In the following example, the graphic control characters (GS, US, ESC, etc.) are assumed to have certain meanings for the terminal. A different terminal may require the use of different control characters to perform a similar functions.

The example shows the use of the graphics instructions described on the preceding pages. This program prints a message, plots a curve with axes, puts the cross hair on the screen, waits for the user to position the cross hair and press a key and carriage return, and then displays the character entered and x,y coordinates of the cross-hair position. You may then end the program or start it again.

Example *(continued)*

```
     GTEST    PROGRAM     START
     START    EQU         *
1             PRINTEXT 'GRAPHICS TEST PROGRAM PRESS ENTER @'
              READTEXT    TEXT1
2             CONCAT   TEXT1,ESC,RESET
3             CONCAT   TEXT1,FF
4             PRINTEXT TEXT1,XLATE=NO
5             STIMER   1000,WAIT
6             CONCAT   TEXT1,GS,RESET
7             SCREEN   TEXT1,520,300,CONCAT=YES
8             CONCAT   TEXT1,US
9             PRINTEXT TEXT1,XLATE=NO
              PRINTEXT    TEXT3
10            YTPLOT   YDATA,X1,PCB,NPTS,1
11            XYPLOT   YAXISX,YAXISY,PCB,TWO
              XYPLOT      XAXISX,XAXISY,PCB,TWO
12            PLOTGIN  X,Y,CHAR,PCB
13            PRINTEXT TEXT4
              PRINTEXT CHAR,XLATE=NO
              PRINTEXT    TEXT5
              PRINTNUM    X,2
14            QUESTION TEXT6,NO=START
              PROGSTOP
     TEXT1    TEXT        LENGTH=30
     TEXT3    TEXT        'X-AXIS LABEL'
     TEXT4    TEXT        '@CHARACTER STRUCK WAS '
     TEXT5    TEXT        '@X,Y COORDINATES ='
     TEXT6    TEXT        '@END PROG (Y/N)? '
              DATA        X'0201'
     CHAR     DATA        F'0'
     YDATA    DATA        F'0'
              DATA        F'1'
              DATA        F'0'
              DATA        F'2'
              DATA        F'0'
              DATA        F'1'
              DATA        F'-2'
              DATA        F'-1'
     X1       DATA        F'0'
     NPTS     DATA        F'8'
     YAXISX   DATA        2F'0'
     YAXISY   DATA        F'-5'
              DATA        F'5'
```

Example *(continued)*

```
XAXISX   DATA      F'0'
         DATA      F'10'
XAXISY   DATA      2F'0'
TWO      DATA      F'2'
PCB      DATA      F'500'
         DATA      F'1000'
         DATA      F'0'
         DATA      F'10'
         DATA      F'100'
         DATA      F'600'
         DATA      F'-5'
         DATA      F'5'
X        DATA      F'0'
Y        DATA      F'0'
         ENDPROG
         END
```

1 Print a message.

2 Reset the text string character count and put the ESC code into TEXT1.

3 Put the FF character into TEXT1.

4 Erase the screen and send the alpha cursor to the home position (upper left corner).

5 Delay for a second to allow the erase sequence to complete.

6 Reset the text string again and insert the graph mode character (GS) to the text string.

7 Form the 4 characters required to draw a dark vector to the screen address (520,300). The 4 characters represent the Hi Y, Lo Y, Hi X, and Lo X values.

8 Write an axis label at this position by returning to alpha mode (US).

9 Perform the full operation. Prevent conversion of data (XLATE=NO), as it is already in ASCII.

10 Plot the data, YDATA (8 points). The plot area and coordinates are given by the 8 words at the label PCB. The plot area in screen addresses is 500 to 1000 in the x-direction (horizontal) and 100 to 600 in the y-direction (vertical). The corresponding plot area in the user's coordinates is 0 to 10 in the x-direction and -5 to 5 in the y-direction.

11 Draw the X and Y axes with this and the next instruction. Each of these is simply a 2-point plot, from the origin to the end point.

12 Put the cross-hair cursor on the screen. The operator should position the cursor and enter a character. When the program receives the character, it converts the cursor position to the plot coordinates as specified at PCB, and stores the results at X and Y.

Example *(continued)*

13 Print the results.

14 Ask if the operator wishes to end the program.

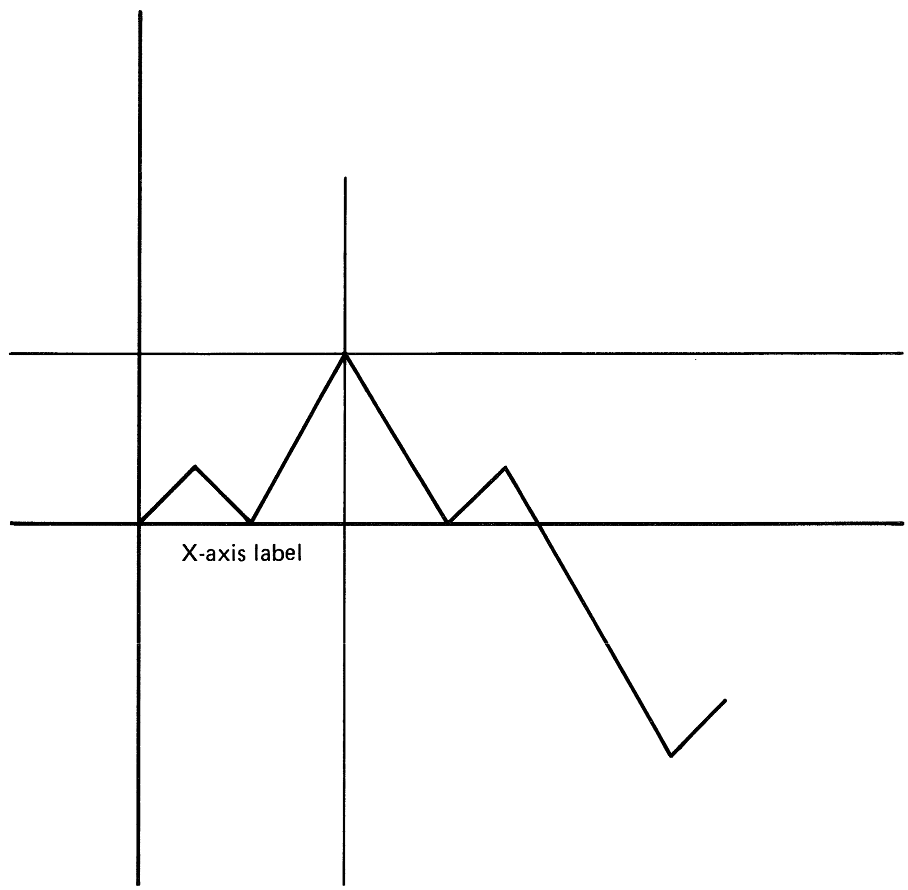


Figure 9. Graphics Program Output. This figure shows the result of the preceding program.

Notes

Chapter 16. Controlling Spooling From A Program

This chapter describes how application programs can control the way their output is handled by spooling under the following headings:

- Determining whether spooling is active
- Preventing spooled printer output
- Separating program output into several spool jobs
- Controlling spool job processing

Figure 11 on page PG-243 depicts the decisions made by the EDL application programmer in designing an application which generates reports.

Determining Whether Spooling Is Active

An EDL application might be such that it should not be run unless spooling has been activated (or deactivated). Such an application can determine if spooling is active and use that information to instruct the operator to activate or deactivate spooling. An application program can also decide whether or not to print a spool-control record, depending on whether or not spooling is activated.

Determining Whether Spooling Is Active *(continued)*

The following EDL coding example shows how an application program can determine if the spooling facility has been activated:

```
MOVE  #2,$CVTSPL,FKEY=0             GET $IOSPTBL ADDRESS
IF    #2,NE,0                       IF IOSPOOL IN SYSTEM
  MOVE  #2,(+$IOSPSPM,#2),FKEY=0      GET $SPM ADDRESS
ENDIF                               ENDIF
IF    #2,NE,0                       IF SPOOLING ACTIVATED
  .
  .
ENDIF
.
.
COPY  PROGEQU                       COPY CVG EQUATES
COPY  $IOSPTBL                      COPY SPOOLING TABLE
```

Figure 10. Determining if Spooling is Active

High-level language programs can call this type of EDL subroutine to determine if spooling is active.

Preventing Spooled Printer Output

An EDL application program can prevent its output from being spooled by coding a parameter on the ENQT command. The parameter is coded as follows:

```
ENQT  SPOOL=NO
```

This causes the printer to be enqueued directly, when available, and prevents output spooling. The SPOOL= parameter is ignored if coded on an ENQT to a device not designated as a spool device or if spooling is not active.

The default is ENQT SPOOL=YES. This allows output spooling.

Note: ENQT SPOOL=NO without the BUSY= operand coded causes the program to wait if a spool writer is started to the device, even if the writer is temporarily stopped. The writer must be terminated to free the device.

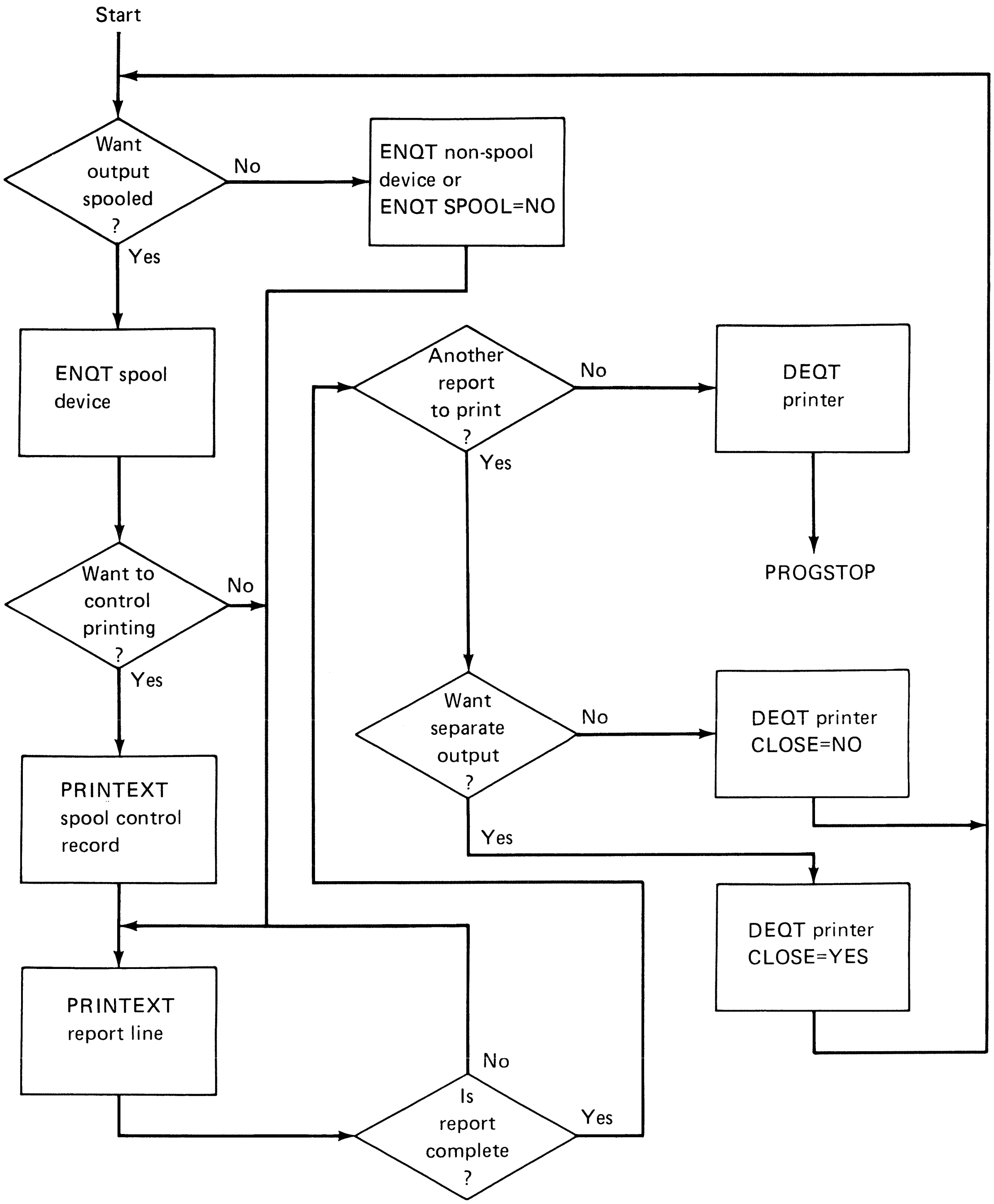


Figure 11. Program-Controllable Spooling Flow Chart

Controlling Spooling From A Program

Separating Program Output into Several Spool Jobs

Spooling treats all spooled printer output of a program to the same printer as a single *spool job*, a group of records printed together. However, a program can produce more than one spool job. To do so, code the DEQT instruction as follows after you create each spool job.

```
DEQT   CLOSE=YES
```

Each time you issue this instruction, the spool job just created becomes ready for printing.

A subsequent ENQT to the same printer indicates the start of a new spool job.

If a program contains more than one task that creates a spool job, you must execute the tasks serially. You must execute a DEQT CLOSE=YES instruction to close each spool job before you execute the next task.

If two tasks spool to the same spool job and one task issues a DEQT CLOSE=YES instruction before the other task has completed spooling, unpredictable results can occur.

The CLOSE= parameter is ignored if coded on a DEQT to a device not designated as a spool device or if spooling is not active.

Note: DEQT CLOSE=NO is the default. It causes any later output of the program directed to the same printer to be concatenated to the output already spooled.

Controlling Spool Job Processing

The application program can control the printing and disposition of its spooled output. This is accomplished by means of a spool-control record.

The Spool-Control Record

The spool-control record, if used, consists of a special print record which must be the first item printed by the program after the device is enqueued.

The spool-control record allows the application program to specify:

- Whether or not the spool job is to be held and not printed.
- Whether or not the spool job is to be kept after printing.
- The type of forms to be used to print the output.
- The number of copies to be printed.
- The separator page heading to be printed.
- Whether forms alignment should be done.

The spool-control record applies only to the spool job which follows it. Thus, if a program creates more than one spool job, and is to control the printing and disposition of each spool job, each spool job must have its own spool-control record.

Note: The $S ALT command, if used by the operator, overrides the spool-control record.

The format of the spool-control record is as follows:

Position	Contents
1-8	***SPOOL
9	blank
10-12	Number of copies to print (1-127)
13	blank
14	Hold disposition (Y=yes, N=no)
15	blank
16	Keep disposition (Y=yes, N=no)
17	blank
18-21	Forms type
22	blank
23-30	Report identification
31	blank
32	Forms aligmment (Y=yes, N=no)

If you use the spool-control record, use care to specify the fields exactly as shown. The fields with a Y/N option default to N. If any character other than a Y or N is entered, the default will be used.

Note: Do not generate the spool-control record in an application program unless spooling has been activated. If spooling is not active, the line is printed as ordinary text to the printer (see "Determining Whether Spooling Is Active" on page PG-241 for a description of how an application program can determine if the spooling facility is active).

Example

The following program uses the spool-control record to create 10 copies with report identification SPOOLPRG, hold and keep disposition in effect, specify forms type ABCD, and specify no forms alignment. The report printed consists of two messages.

```
SPOOL   PROGRAM   START
START   EQU       *
        ENQT
        PRINTEXT  '***SPOOL 010 Y Y ABCD SPOOLPRG N'
        PRINTEXT  '@MESSAGE 1'
        PRINTEXT  '@MESSAGE 2'
        DEQT
        PROGSTOP
        ENDPROG
        END
```

Programming Considerations

You should be aware of the following things when you spool output:

- A program can activate spooling by loading the spool manager into storage. You can start a spool writer automatically when the spool session begins.
- Use the forms alignment option for any report that must be registered precisely on a form.
- The GETSTORE and PUTSTORE functions of the TERMCTRL instruction have no effect on a spool device.
- If the spool data set is out of space or the control blocks required by spooling are not available, any program that is spooling waits until space is available.
- If you direct a spool job to a 4975 printer, the program should issue a TERMCTRL SET,RESTORE instruction at the beginning and end of the job.
- If you define an IOCB for a printer and you specify operands on the IOCB, code a TERMCTRL DISPLAY instruction after you issue an ENQT instruction and before you issue any subsequent ENQTs to the same IOCB.
- If you designate $SYSLOG as a spool device, a program check will not print.

Chapter 17. Creating, Storing, and Retrieving Program Messages

When designing EDL programs, you can save storage space or coding time by placing prompt messages and other message text in a separate message data set. EDL instructions enable your program to retrieve the appropriate message text when the program executes.

By storing messages in a data set, you can change the text of a message without having to alter and recompile each program that uses that message.

You can store program messages in two ways. You can store them on disk or diskette. You can also store them as a module that you can link-edit with a program.

Creating and using your own program messages involves the following steps:

1. Creating a data set for your source messages

2. Entering your source messages

3. Formatting and storing your source messages using the message utility, $MSGUT1

4. Retrieving program messages using the COMP statement and the MESSAGE, GETVALUE, QUESTION or READTEXT instructions

The following sections describe how to create, store, and retrieve program messages.

Creating, Storing, and Retrieving Program Messages

Creating a Data Set for Source Messages

You create a data set for source messages with the text editor described in Chapter 3, "Entering a Source Program" on page PG-59. You can create one or more source message data sets and can store them on any volume. Messages can be simple statements or questions, or they can include variable fields which are filled with parameters supplied by your program.

To enter your source messages, observe the following 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 each message 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.
- Code source messages a maximum length of 253 bytes long. You can calculate the length of a message by adding one byte for each character in the text and one byte for each variable field.
- Continue a message on a new line by coding any non-blank character in column 72. Begin the continued line in the first column.

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 your source messages with the $MSGUT1 utility, you should add any new messages you have to the end of the source message data set. If you no longer need a certain message, you should leave it in the source message data set or replace it with a new message to preserve the numbering scheme.

Coding Messages with Variable Fields

To construct a message that can return information supplied or generated by your program, 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, if you want to construct a message that tells a program operator how many records are in a particular data set on a particular volume, you could 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.

Creating a Data Set for Source Messages *(continued)*

Note: To print or display a message with variable fields, you must have included the FULLMSG module in your system during system generation.

The variable fields are set 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 eight variable fields within a single message.

As shown in the previous example, all variable fields must also contain a **control character** that describes the type of parameter your program will pass to the variable field. *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 valid control characters and their descriptions:

C Character data. Specify a length for the data by coding a value from 1 to 253 before the 'C' (for example, <<NAME>8C>). There is no default.

T Text. No length is necessary. (The system derives the length from the TEXT statement.)

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 instruction. The following example shows a message instruction with the parameter list operand (PARMS=):

```
MSG       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 instruction will retrieve message number 2. The source message for message number 2 appears as follows:

```
<<DATA SET NAME>T> ON <<VOLUME>T> IS ONLY <<SIZE>S> RECORDS./*
```

The system places the first parameter (DSNAME) in the first variable field, the second parameter (VOLUME) in the second field, and the third parameter (SIZE) in the third field.

Creating a Data Set for Source Messages *(continued)*

You may, however, want to alter or reword the message in the previous example. To change the order of the variable fields in your source message without changing the order of the parameter list in your program, you can 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 a parameter list.

In the following example, the order of the variable fields in message number 2 has been switched, but a number following the control character points to the correct parameter for the variable field:

```
THERE ARE ONLY <<SIZE>S3> RECORDS IN <<DATA SET NAME>T1> ON   C

<<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 sample of a source message data set. The data set is named SOURCE on volume EDX40.

```
//THIS IS A COMMENT                                                  //+
DO YOU WANT TO ENTER A NUMBER? /*
ENTER <<TYPE OF VALUE>T> VALUE LESS THAN <<VALUE>S>./*
THE PROGRAM HAS PROCESSED THE INPUT DATA./*
ENTER YOUR <<FIRST/LAST/FULL NAME>10C>./*
//THIS IS ANOTHER COMMENT.  //                                         +
ALL INPUT DATA HAS BEEN RECEIVED./*
THE VALUE YOU ENTERED IS: <<VALUE>S1> /*
THE DATA YOU ENTERED IS: <<DATA>T> /*
THE DEVICE <<ID>H1> AT ADDRESS <<DEVICE ADDRESS>H2> IS IN USE./*
THIS MESSAGE WILL BE CONTINUED @ ON THE NEXT LINE./*
```

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 in another data set or module that you specify. (Refer to the *Operator Commands and Utilities Reference* for a detailed explanation of how to use the message utility.)

Each time you add new messages to the source message data set, you must reformat the data set with $MSGUT1.

The $MSGUT1 utility allows you to:

- Format a source message data set and store the formatted messages on disk or diskette.
- Format a source message data set as a module that you link-edit with a program. Use this option for systems without disk or diskette storage or to improve performance.
- Obtain a hard-copy listing of the messages contained in a specific source message data set.

Before you load the $MSGUT1 utility, you must allocate a work file. You can use the AL command of the $DISKUT1 utility to allocate the work file. Allocate a data-type data set large enough to hold the source message data set (one record for every source message).

When you load $MSGUT1, the utility prompts you for the name and volume of the work file as follows:

```
WORKFILE (NAME,VOLUME):
```

Respond with the data set name and volume that you allocated with the $DISKUT1 utility.

Example 1

In the following example, $MSGUT1 formats the source message data SOURCE shown in the previous section. The example uses the DSK option and stores the formatted messages in the data set MESSAGE on volume EDX40.

```
COMMAND (?): DSK
MESSAGE SOURCE DATA SET (NAME,VOLUME): SOURCE,EDX40
DISK RESIDENT DATA SET (NAME,VOLUME): MESSAGE,EDX40
START OF DISK MESSAGE PROCESSING BEGINS
```

When the utility finishes formatting and storing the messages, it returns the following message:

```
DISK RESIDENT MESSAGES STORED IN MESSAGE,EDX40
```

Example 2

The following example uses the STG option and stores the module in data set MSG on volume EDX003.

```
COMMAND (?): STG
MESSAGE SOURCE DATA SET (NAME,VOLUME): MSGSRC,EDX003
STORAGE RESIDENT MODULE (NAME,VOLUME): MSG,EDX003
START OF STORAGE MESSAGE PROCESSING
```

Formatting and Storing Source Messages (using $MSGUT1) *(continued)*

When the utility finishes formatting and storing the messages, it returns the following message:

```
STORAGE RESIDENT MODULE STORED IN MSG,EDX003
```

If the $MSGUT1 utility encounters errors, it prints an error message on the system printer.

Retrieving 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. (Refer to the *Language Reference* for a full description of these instructions and how to code them to retrieve messages.)

The system retrieves program messages from the data set or module that you created with $MSGUT1. If you stored your formatted messages on disk or diskette, you must code the name of the data set that contains the messages and the volume it resides on in the PROGRAM statement for your program.

If you formatted the messages as a module, you must link-edit your program with the module.

Defining the Location of a Message Data Set

The COMP statement defines the location of a message data set or the name you assigned the module when you used the STG option of the $MSGUT1 utility. To retrieve a message, the MESSAGE, GETVALUE, QUESTION, and READTEXT instructions must refer to the label of a COMP statement. More than one instruction can refer to the same COMP statement. You must code a separate statement, however, for each message data set your program uses.

If your messages are in a module, you must code the name of the module. If your message data resides on disk or diskette, you must indicate the data set in the PROGRAM statement. You indicate the correct data set by specifying its position in the data set list.

In addition to coding the location of the message data set, you must also code a four-character prefix. The system prints this prefix and the number of the message you retrieved if you specify (MSGID=YES) on the MESSAGE, GETVALUE, QUESTION, or READTEXT instructions.

The following example shows a COMP statement that refers to the second data set on the PROGRAM statement. DS2 points to data set MESSAGE on volume EDX40.

```
MESSAGE      PROGRAM      START,DS=(DATA,(MESSAGE,EDX40))
                .
                .
             PROGSTOP
DISKMSG      COMP         'ERRS',DS2,TYPE=DSK
```

The following example shows a COMP statement that refers to a module that contains messages.

```
MESSAGE      PROGRAM      START
                .
                .
             PROGSTOP
STGMSG       COMP         'ERRS',MSG,TYPE=STG
```

The MESSAGE instruction

The MESSAGE instruction retrieves a message from a data set on disk, diskette, or from a module. Then the instruction prints or displays the message. You must code the number of the message you want displayed or printed and the label of the COMP statement that gives the location of the message (COMP=).

You can pass parameters to variable fields in a message by coding the parameters on the PARMS= operand of the instruction. If you code MSGID=YES, the system prints or displays the number of the message and the four-character prefix you coded on the COMP statement in front of the message text.

In the following example, the MESSAGE instruction retrieves the third message in a message data set and passes the parameter PART# to the message. The COMP statement defines the message data set as the first data set in the PROGRAM statement list.

```
STOCK        PROGRAM      START,DS=(PARTS,DATA)
             MESSAGE      3,COMP=PARTS,PARMS=PART#,MSGID=YES
                .
                .
             PROGSTOP
PARTS        COMP         'PART',DS1,TYPE=DSK
PART#        DC           F'56'
```

In the following example, the MESSAGE instruction retrieves the second message in a module that has been link-edited with the program and passes the message the parameter PART#. The COMP statement defines the message data set as module MSG.

```
STOCK        PROGRAM      START
             MESSAGE      2,COMP=PARTS,PARMS=PART#,MSGID=YES
                .
                .
             PROGSTOP
PARTS        COMP         'PART',MSG,TYPE=STG
PART#        DC           F'43'
```

The GETVALUE, QUESTION, and READTEXT Instructions

Instead of coding prompt messages on the GETVALUE, QUESTION, and READTEXT instructions, you can retrieve prompt messages from a message data set or module. You code the number of the message you want to retrieve for the second operand of the GETVALUE and READTEXT instructions and the first operand of the QUESTION instruction. In addition, you must code the label of the COMP statement that gives the location of the message (COMP=).

You can pass parameters to variable fields in a message by coding the parameters on the PARMS= operand of the instruction. By coding MSGID=YES, the system prints or displays the number of the message and the four-character name you coded on the COMP statement at the front of the message text.

In the following example, the GETVALUE instruction retrieves the fifth message from a module, called MSGTEXT, that has been link-edited with your program. The instruction also passes the message the parameters VALUE and SIZE to the message.

```
           GETVALUE   INPUT,5,COMP=PROMPT,PARMS=(VALUE,SIZE)
               .
               .
               .
           PROGSTOP
PROMPT     COMP       'TASK',MSGTEXT,TYPE=STG
VALUE      TEXT       'AN INTEGER'
SIZE       DC         F'75'
```

In the following example, the GETVALUE instruction retrieves the ninth message from a data set on disk or diskette. The instruction passes the message the parameters VALUE and SIZE.

```
BEGIN      PROGRAM    START,DS=MSGS
               .
               .
           GETVALUE   INPUT,9,COMP=PROMPT,PARMS=(VALUE,SIZE)
               .
               .
               .
           PROGSTOP
PROMPT     COMP       'TASK',DS1,TYPE=DSK
VALUE      TEXT       'AN INTEGER'
SIZE       DC         F'75'
```

Sample Program

The following sample program retrieves five program messages from a disk data set formatted in the previous section. (See "Example 1" on page PG-251.) The name of the data set is MESSAGE and it resides on EDX40.

```
1  MESSAGE   PROGRAM    START,DS=((MESSAGE,EDX40))
2  START     QUESTION   1,NO=NAME,SKIP=1,COMP=DISKMSG
3            GETVALUE   A,2,SKIP=1,COMP=DISKMSG,PARMS=(P1,P2)
             PRINTEXT   '@THE NUMBER IS: '
4            PRINTNUM   A,SKIP=1
5  NAME      READTEXT   B,+MSG4,SKIP=1,COMP=DISKMSG,PARMS=TXT
             PRINTEXT   '@THE DATA ENTERED IS: '
6            PRINTEXT   B,SKIP=1
7            MESSAGE    +MSG6,COMP=DISKMSG,SKIP=2,PARMS=A,          C
                 MSGID=YES
8            MESSAGE    +MSG7,COMP=DISKMSG,SKIP=2,PARMS=B,          C
                 MSGID=YES
             MESSAGE    +MSG9,COMP=DISKMSG,SKIP=2,PARMS=B,          C
                 MSGID=YES
             PROGSTOP
9  MSG4      EQU        4
   MSG6      EQU        6
   MSG7      EQU        7
   MSG9      EQU        9
10 DISKMSG   COMP       'SRCE',DS1,TYPE=DSK
11 A         DATA       F'0'
   B         TEXT       LENGTH=40
   P1        TEXT       'AN INTEGER'
   P2        DATA       F'10'
   TXT       DATA       CL10'LAST NAME '
             ENDPROG
             END
```

1 Begin the program and identify the data set name and volume of the message data set (MESSAGE on volume EDX40).

2 Display the prompt message DO YOU WANT TO ENTER A NUMBER? The first operand (1) identifies the message as the first message in the data set MESSAGE. The COMP= operand refers to a COMP statement labeled DISKMSG. If the operator enters *Y*, the next sequential instruction, the GETVALUE instruction, executes. If the operator enters *N*, control passes to the label NAME.

3 Use the second message in the message data set as a prompt message. The instruction retrieves the prompt message and inserts parameters P1 and P2 into the message. The operator receives the prompt message **ENTER AN INTEGER VALUE LESS THAN 10**.

4 Print the number the operator enters.

Sample Program *(continued)*

5 Retrieve the fourth message (because MSG1 is equated to 4) from the message data set and inserts parameter TXT into the message. The operator receives the prompt message **ENTER YOUR LAST NAME**.

6 Print the name the operator enters.

7 Print or display the sixth message (because MSG6 is equated to 6) from the message data set. The COMP= operand refers to the COMP statement labelled DISKMSG. The instruction uses the integer value the operator entered as the parameter for the message. If the operator entered a 6, for example, the system would print or display: **THE VALUE YOU ENTERED IS 6**.

8 Print or display the seventh message (because MSG7 is equated to 7) from the message data set. The COMP= operand refers to the COMP statement labelled DISKMSG. The instruction uses the last name the operator entered as the parameter for the message. If the operator entered the name FRENCH, for example, the system would print or display: **SRCE0007 THE DATA YOU ENTERED IS FRENCH**.

9 Equate MSG4 to the fourth message in the message data set.

10 Define the message data set as the first data set on the PROGRAM statement. Identify the data set as a disk- or diskette-resident data set (TYPE=DSK). **SRCE** is the prefix that would appear if you coded MSGID=YES on a QUESTION, PRINTEXT, GETVALUE, or READTEXT instruction.

11 Define a parameter (used by the first MESSAGE instruction).

The program uses the following source message data set:

```
//THIS IS A COMMENT                                                     //+
DO YOU WANT TO ENTER A NUMBER? /*
ENTER <<TYPE OF VALUE>T> VALUE LESS THAN <<VALUE>S>./*
THE PROGRAM HAS PROCESSED THE INPUT DATA./*
ENTER YOUR <<FIRST/LAST/FULL NAME>10C>./*
//THIS IS ANOTHER COMMENT.  //                                            +
ALL INPUT DATA HAS BEEN RECEIVED./*
THE VALUE YOU ENTERED IS: <<VALUE>S1> /*
THE DATA YOU ENTERED IS: <<DATA>T> /*
THE DEVICE <<ID>H1> AT ADDRESS <<DEVICE ADDRESS>H2> IS IN USE./*
THIS MESSAGE WILL BE CONTINUED @ ON THE NEXT LINE./*
```

The program might produce output like the following:

```
DO YOU WANT TO ENTER A NUMBER? Y

ENTER AN INTEGER VALUE LESS THAN      10: 4

THE NUMBER IS:      4
ENTER YOUR LAST NAME : MEGATH

THE DATA ENTERED IS: MEGATH

SRCE0006 THE VALUE YOU ENTERED IS:     4

SRCE0007 THE DATA YOU ENTERED IS: MEGATH

SRCE0009 THIS MESSAGE WILL BE CONTINUED
 ON THE NEXT LINE.
```

Notes

Chapter 18. Queue Processing

You can use the queue processing instructions of EDL to store and retrieve large amounts of data. You can retrieve data from a queue on either a first-in-first-out or last-in-last-out basis.

Defining a Queue

To define a queue, use the DEFINEQ statement. The following DEFINEQ statement defines a queue with ten queue elements. A *queue element* is either an address or data that you want to store.

```
MSGQ    DEFINEQ   COUNT=10
```

The queue called MSGQ can contain ten one-word addresses or one-word data items.

If you want to store data items that are longer than one word, code the SIZE operand as follows:

```
QUEUE   DEFINEQ   COUNT=15,SIZE=30
```

The queue called QUEUE can contain 15 thirty-byte queue elements.

Putting Data into a Queue

To put data into a queue, use the NEXTQ instructions as follows:

```
         NEXTQ    MSGQ,ADDR
         .
         .
ADDR     DATA     F'0'
```

The instruction puts ADDR into the queue called MSGQ. ADDR can contain either one word of data or an address.

To put more than one word of data into a queue, use the FIRSTQ instructions to find the address of the first storage area into which data can be moved.

```
         FIRSTQ   QUEUE,#1
         .
         .
QUEUE    DEFINEQ COUNT=15,SIZE=20
```

The instruction puts into register 1 the address of the first storage area into which you can move twenty bytes of data.

You could use the following instructions to prompt the operator for data and store the response in QUEUE:

```
READTEXT  ELEMENT,'ENTER YOUR NAME: '
MOVE      (0,#1),ELEMENT,(20,BYTE)
```

The READTEXT instruction prompts the operator and places the response in ELEMENT. The MOVE instruction moves the response to the address retrieved by the FIRSTQ instruction.

Retrieving Data from a Queue

To retrieve data from a queue, use either the FIRSTQ or LASTQ instruction.

Use the FIRSTQ instruction to retrieve the oldest entry from a queue. The following example

```
FIRSTQ   QUEUE,#2
```

puts into register 2 the address of the oldest element in the queue called QUEUE.

Use the LASTQ instruction to retrieve the newest entry from a queue. The following example

```
LASTQ   QUEUE,ADDR
```

puts into ADDR the address of the oldest element in the queue called QUEUE.

Retrieving Data from a Queue *(continued)*

To transfer control if the queue becomes empty, code the EMPTY operand as follows:

```
         FIRSTQ QUEUE,ADDR,EMPTY=MT
         .
         .
MT       EQU    *
         .
         .
ADDR     DATA   F
```

The instruction retrieves an element from the queue called QUEUE, puts the address of the element in ADDR, and causes a branch to MT if no more elements exist in the queue.

Example

The following example prompts the operator for 20 characters of data, stores the data in one queue, moves the addresses of the elements to another queue, and prints the elements on a first-in-first-out (FIFO) basis.

```
   QTEST    PROGRAM  START
   START    EQU      *
            DO       10,TIMES
1           FIRSTQ   QUEUE1,#1
2           READTEXT MSG,'ENTER UP TO 20 CHARACTERS: '
3           MOVE     (0,#1),MSG,(20,BYTE)
4           NEXTQ    QUEUE2,#1,FULL=FULLQ
            ENDDO
            GOTO       PRINT
   FULLQ    EQU        *
            PRINTEXT   '@QUEUE2 FULL.'
   PRINT    EQU        *
            DO         10,TIMES
5           FIRSTQ     QUEUE1,#1,EMPTY=DONE
6           MOVE       MSG,(0,#1),(20,BYTE)
7           PRINTEXT   MSG,SKIP=1
8           NEXTQ      QUEUE1,#1
            ENDDO
   DONE     PROGSTOP
9  QUEUE1   DEFINEQ    COUNT=10,SIZE=20
10 QUEUE2   DEFINEQ    COUNT=10
   MSG      TEXT       LENGTH=20
            ENDPROG
            END
```

Example *(continued)*

1 Put the address of the oldest element into register 1.

2 Prompt the operator for twenty characters of data. Put the prompt in MSG.

3 Move the operator's response into QUEUE1, to the address retrieved by the FIRSTQ instruction.

4 Store in QUEUE2 the address where the response was stored in QUEUE1.

5 Retrieve the oldest element from QUEUE1 and put the address of the data into register 1.

6 Move twenty bytes from the address pointed to by register 1 to MSG.

7 Print the data, skipping a line between each data item (SKIP=1).

8 Put back into QUEUE1 the element retrieved by the FIRSTQ instruction.

9 Define a queue large enough to accommodate ten 20-character data items.

10 Define a queue large enough to accommodate ten 1-word data items or addresses.

Appendix A. Tape Labels

The following is the layout of the VOL1 label:

Field Name	Bytes	Initialized Contents
Label identifier	3	VOL
Volume label number	1	1
Volume serial	6	XXXXXX
Volume security	1	0
Data file directory	10	blanks
Reserved	10	blanks
Reserved	10	VOL
Owner name	10	NAME
Reserved	29	blanks

The following is the layout of the HDR1 label:

Field Name	Bytes	Initialized Contents
Label identifier	3	HDR
File label number	1	1
File identifier (DSN)	17*	Data set name (DSN)
File serial number	6	XXXXXX
Volume sequence number	4	0001
File sequence number	4	00NN
Generation number	4	blanks
Generation version number	2	blanks
Creation date	6	YYDDD
Expiration date	6	YYDDD
File security	1	0
Block count	6	000000
System code	13	IBMEDX1
Reserved	7	blanks

* EDX supports an 8-byte nonblank data set name (DSN). EDX ignores the last 9 bytes of the DSN.

Notes

Appendix B. Interrupt Processing

Interrupts apply to the interaction between a program and a terminal operator. For example, a program can wait for an interrupt, such as an operator response to a prompt, or a terminal operator can cause an interrupt by pressing a Program Function key.

When an interrupt occurs, if it is completing an outstanding operation, control is returned to the next sequential instruction if there are no errors. If the interrupt was unsolicited (caused by the attention key or a PF key), then either the system or user ATTNLIST begins executing as an asynchronous task competing for system resources.

Interrupt Keys

The keys that can cause interrupts are the attention key, Program Function (PF) keys and the enter key.

The Attention Key

When the attention key is recognized, the greater than symbol (>) is displayed and the operator can enter either a system function code (for example, $L) or a program function code defined in an ATTNLIST.

The attention key on the 4978 and 4979 is the key marked ATTN. For teletype terminals, the ESC (escape) key is usually the attention key. For the 3101 Display Terminal, the PF8 key is the default attention key.

Interrupt Keys *(continued)*

Program Function (PF) Keys

Any program function key on the 4978/4979 and 3101 is recognized by the attention list code $PF (except for a PF key defined as the attention key). In addition, individual keys can be separately recognized by $PF1 to $PF254. You can provide separate entry points to the application code for particular keys, or a single entry point for all keys or a group of keys for rapid response.

The order of the PF keys in the attention list is significant because it defines the entry points to the application code. For example:

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

causes the program to be entered at ENT3 for all PF keys except PF1 and PF5.

On the 4978/4979, pressing the PF6 key causes the screen image to be printed on any designated hard-copy terminal (unless that terminal is a spool device and spool is loaded). This is not true for PF6 on the 3101.

The 3101 keyboard has eight PF keys. EDX supports these keys when the 3101 is operated in both character and block mode. To use the PF keys on the 3101, hold down the ALT key (on the lower right-hand side of the keyboard) while you press the appropriate numeric key.

Enter Key

The enter key indicates the end of typed input, for example, the end of the operator input for a READTEXT instruction. You also use it in conjunction with the WAIT KEY instruction.

On the 4978 and 4979 keyboards, the enter key is marked ENTER. For the 3101 in block mode, the SEND key is the enter key. For the 3101 in character mode, the new line key is the enter key.

Instructions that Process Interrupts

Instructions that process interrupts are READTEXT, GETVALUE, WAIT KEY and ATTNLIST.

The READTEXT and GETVALUE Instructions

In many cases a program needs to wait for an interrupt, such as an operator response to a request for input. This program-wait capability is provided automatically by the READTEXT and GETVALUE instructions. These instructions have an "implied wait."" They wait for the terminal operator to enter data and press the enter key.

The WAIT KEY Instruction

An application program can wait at any point for a 4978/4979 or 3101 terminal operator to press the enter or one of the PF keys. This is done by issuing the WAIT KEY instruction.

When the enter or a PF key is pressed, the program resumes operation, and the key is identified to the program in the second task code word at taskname+2. The code value for the enter key is 0. The value for a PF key is the integer corresponding to the assigned function code; 1 for PF1, 2 for PF2, and so on.

The PF keys do not initiate attention list processing during execution of the WAIT KEY instruction. They only cause the WAIT KEY instruction to terminate, allowing subsequent instructions to be executed.

The ATTNLIST Instruction

The ATTNLIST instruction provides entry to interrupt processing routines. When a PF key is pressed, the ATTNLIST task for that key gets control if ATTNLIST was coded in the application program. If ATTNLIST was not coded, the system search for a PF key match fails and the message "FUNCTION NOT DEFINED" is displayed on the screen. Except for the 4978/4979 hard-copy print key (normally PF6), the 4978 attention key (normally PF0) and the 3101 attention key (normally PF8), the PF keys are always matched against user-written ATTNLIST(s) as described above.

When the attention key on a terminal is pressed, the system prompts the operator for a command. This command is first matched against the system ATTNLIST and then against user-written ATTNLIST(s).

If the command matches the system ATTNLIST, appropriate system action is taken (for example, $D or $L) unless the task is busy. If the command entered was $C, $VARYON or $VARYOFF and this task is busy, the message "> NOT ACKNOWLEDGED" is displayed; when the task is completed, $C, $VARYON or $VARYOFF is then executed. If the command entered was $P or $D and this task is busy, the command is ignored.

If the command matches a user-written ATTNLIST, the corresponding ATTNLIST task gets control. The appropriate application program attention routine then runs under this task. If the attention key invoked the ATTNLIST and the task is already busy, the message "> NOT ACKNOWLEDGED" is displayed on the terminal.

If there is no match against any ATTNLIST, the message "FUNCTION NOT DEFINED" is displayed.

When the ATTNLIST task for a PF key gets control, the code for that key is placed in the second word of the ATTNLIST task control block. You can obtain the code for an interrupting key by coding the TCBGET instruction.

Advance Input

As a terminal user, your interaction with an application or utility program is generally conducted through prompts which request you to enter data. Once you have become familiar with the dialogue sequence, however, prompting becomes less necessary. The READTEXT and GETVALUE instructions include a conditional prompting option which enables you to enter data in advance and thereby inhibit the associated prompts.

Advance input is accomplished by entering more data on a line than has been requested by the program. Subsequent input instructions specifying PROMPT=COND will read data from the remainder of the buffered line, and issue a prompt only when the pre-entered data has been exhausted. If you specify PROMPT=UNCOND with an input instruction, an associated prompt is issued and the system waits for input. The prompt causes, as does every output instruction, cancellation of any outstanding advance input.

Appendix C. Static Screens and Device Considerations

EDX terminal support enables some degree of device independence between the 4978/4979 Display Station and 3101 Display Terminal in block mode. This device independence is achieved by using the $IMAGE subroutines and certain parameters of some EDL instructions, namely READTEXT, PRINTEXT and TERMCTRL. This type of device independence applies only when the terminals are using static screens.

This chapter first discusses static screens, including how to design static screen applications for terminal independence. The $IMAGE subroutines are described and an example of using them is shown. Two sample programs are provided, one for the 4978/4979 and one for the 3101 in block mode.

The final section of the chapter deals with characteristics unique to certain EDX terminals, namely teletypewriter terminals, ACCA terminals, and EDX terminals that are other processors.

A Description of Static Screens

A static screen is 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 the terminal operator. The screen is treated as a page of information.

The object of static screen management is to provide the application program with complete control over the screen image, and to allow the terminal operator to modify an entire screen image before data entry. Static screens are therefore distinguished from roll screens in the following ways:

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

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

- The cursor position, relative to the logical screen margins, can be determined by the application program using the RDCURSOR instruction.

- Input operations directed to static screens normally do not cause a task suspension wait for the enter key; they are executed immediately. This allows the program to read selected fields from the screen after the entire display has been modified by the operator. Operator/program signaling is implemented using the Program Function keys and the WAIT KEY instruction.

- To allow convenient operator/program interaction, QUESTION, READTEXT, and GETVALUE instructions which include prompt messages are executed as if they were directed to a roll screen (automatic task suspension for input).

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

The utility program $IMAGE (see *Operator Commands and Utilities Reference*) constructs formatted screen images in a interactive mode and saves them in disk or diskette data sets. The images are retrieved and displayed by application programs through the use of system-provided subroutines called the $IMAGE subroutines. See "Using the $IMAGE Subroutines for Device Independence" on page PG-278 for details.

Defining Logical Screens

A logical screen is a screen defined by margin settings, such as the TOPM, BOTM, LEFTM and RIGHTM parameters. Logical screens can be defined either during system generation (using the TERMINAL statement) or at the time an ENQT instruction is executed (using the IOCB statement).

Using TERMINAL to Define a Logical Screen

The following example of using the TERMINAL statement defines a static screen to be used for data entry and display. Programs can be loaded from the terminal, but the terminal I/O instructions issued will be interpreted for a static screen unless the configuration is changed to roll by an IOCB statement. This is a typical definition for a terminal to be used for data entry.

```
TERM2    TERMINAL  DEVICE=4979,ADDRESS=14,SCREEN=STATIC
```

The next example shows a split screen configuration. The roll screen is the bottom 12 lines of the screen; the top half can be used for other logical screens defined upon execution of ENQT.

```
TERM3     TERMINAL  DEVICE=4978,ADDRESS=24,TOPM=12,NHIST=6
```

The next example defines a roll screen occupying the upper-right quadrant of the screen. In general, logical screens with less than an 80-character line size suffer some performance disadvantages (such as slower erasure) but can be useful for special applications. Note that NHIST is zero here because screen shifting will not be performed; a non-zero value for NHIST would merely cause the history area to be unused.

```
TERM4     TERMINAL  DEVICE=4979,ADDRESS=34,LEFTM=39,                 C
                 BOTM=11,NHIST=0
```

The final example defines a static screen for the 3101 in block mode. A 3101 can have only a single roll or a single static screen. The Multifunction Attachment is used to connect the terminal to the Series/1.

```
TERM5     TERMINAL  DEVICE=ACCA,ADDRESS=59,MODE=3101B,               C
                 SCREEN=STATIC,LMODE=RS422,ADAPTER=MFA
```

Using IOCB and ENQT to Define a Logical Screen

Logical screens can also be defined by the ENQT instruction referencing an IOCB. The IOCB statement is used to define many of the "soft" characteristics of a terminal (such as margins, page size or line length) and to establish the connection between the ENQT and TERMINAL statements at execution time. Using an ENQT instruction which references an IOCB, you can modify the soft characteristics of a specific terminal defined by the TERMINAL statement. The IOCB statement and its operands are fully described in the *Language Reference*.

In the following example, the IOCB labeled TOPHALF defines the top half of the screen (from which the program was loaded) as a static screen. If the terminal were defined as in TERM3 on the previous page, the program could have been loaded by entering $L program-name in the roll screen area (the bottom half of the screen). Since no terminal name is specified on the IOCB statement, the ENQT refers to the loading terminal. The program then might display tabular information on the static screen, execute DEQT and then end. The information displayed on the static screen part of the screen will remain on the screen while the terminal operator performs other operations using the roll screen.

```
DISPLAY    PROGRAM    BEGIN
TOPHALF    IOCB       BOTM=11,SCREEN=STATIC
BEGIN      ENQT       TOPHALF
           .
           .
           .
           DEQT
           PROGSTOP
           ENDPROG
           END
```

The next example shows terminal access by using the symbolic name of the terminal. TERM1, TERM2, TERM3, and TERM4 have all been defined with TERMINAL configuration statements. The use of a static screen ensures that only physical line 0 of each screen will be altered. (LINE=0 for roll screens causes a page eject and erasure of information.)

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

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

Structure of the IOCB

The structure of the IOCB is given in the following table. The structure may change with future versions of the Event Driven Executive.

Field Name	Byte(s)	Contents
Terminal name	0-7	EBCDIC, blank filled
Flags	8	Bit 0 off indicates that the name is the only element of the IOCB. Further information on this field can be found in *Internal Design*.

A Description of Static Screens *(continued)*

Field Name	Byte(s)	Contents
Top of working area	9	Equal to TOPM+NHIST
Top margin	10	TOPM or zero
Bottom margin	11	BOTM, or X'FF' if unspecified
Left margin	12	LEFTM or zero
Page size	13	Equal to X'00' if unspecified
Line size	14-15	Equal to X'7FFF' if unspecified
Current line	16	Initialized to TOPM+NHIST
Current indent	17	Initialized to left margin
Buffer address	18-19	Zero if unspecified

Some Characteristics of the 3101 Display Terminal

Attribute Characters

The 3101 uses attribute characters (or bytes) to define fields on the screen. An attribute byte defines the start of each field and the properties of the field (such as protected/unprotected, high/low intensity). Each attribute byte appears as a protected blank on the screen.

The collection of attribute characters, special sequences required by the terminal, and user data is called a "data stream". Any invalid (unprintable) characters encountered in the data stream will cause the alarm to ring. This condition might occur, for instance, when displaying a non-EBCDIC disk or diskette record.

Transmitting Data from the 3101

On a 3101 static screen, the application program must determine where the output data is positioned, relative to the first position of the screen. For READTEXT operations, modified, data, or all fields are read from the beginning of the screen (regardless of forms control), depending on the TYPE parameter of the READTEXT instruction.

In response to a read request, the 3101 transmits the attribute characters that precede the input field. To suppress the attribute characters from the data stream, the EDX 3101 support removes these special characters and left-justifies the data.

Some Characteristics of the 3101 Display Terminal *(continued)*

A feature is provided that allows an application program to have complete control of the input/output data transmitted to or from the terminal. To do this, the program must build the complete data stream, either in EBCDIC or ASCII codes. The basic terminal I/O support simply handles the transmission of the data stream. Refer to the description of the TERMCTRL SET,STREAM=YES/NO instruction and the XLATE parameter of PRINTEXT/READTEXT instructions in the *Language Reference* when this mode of data transmission is desired.

Screen Formats

A screen format is a representation of the protected fields on a screen. Screen formats and input/output are handled differently on the 4978/4979 and 3101. References to the 3101 Display Terminal in this section mean a 3101 model 2x operating in block mode.

4978/4979 Screen Formats

The format of a 4978/4979 screen is defined as each character is written to the terminal. Fields are defined as follows:

- Each character or group of characters written with PROTECT=YES defines a protected field.
- Each character or group of characters written without PROTECT=YES defines an unprotected field.
- Null characters (X'00') can never be protected, so both protected and unprotected fields can be defined by writing data with interspersed nulls with PROTECT=YES.

Once the fields of a screen have been defined, the 4978/4979 knows internally whether each of the 1920 positions on the screen is protected or unprotected; this is transparent to the user.

On the 4978/4979 there are two ways to write and read unprotected fields. The first is to read/write all the unprotected fields with one input/output operation. All the unprotected fields can be filled with data by one "scatter write" operation (PRINTEXT MODE=LINE). The unprotected fields can be read using one "gather read" operation (READTEXT MODE=LINE). The other way is to read or write individual fields by specifying screen coordinates (the LINE= and SPACES= parameters).

3101 Screen Formats

Like the 4978/4979, the format of a 3101 screen is defined by how the data is written, either protected or unprotected. However, on the 3101, the field definitions are not transparent to the user because attribute bytes separate protected and unprotected fields.

- An attribute byte defines the start of each field and the properties of the field.

- Each field continues until another attribute byte is encountered.
- Each attribute byte occupies one character position on the screen and is displayed as a protected blank preceding the field.
- Attribute bytes are like any other character on the screen in that they can be overwritten by data or another attribute byte. When an attribute byte is overwritten, the screen format can change.

On a 3101 it is not possible to do a scatter write with a PRINTEXT instruction; however, you can specify screen coordinates on output (PRINTEXT LINE=,SPACES=). You can do a gather read by specifying READTEXT MODE=LINE. However, the input of a specific field (by means of READTEXT LINE=,SPACES=) always executes as though LINE=0 and SPACES=0 had been coded.

As a result of these differences between the 4978/4979 and the 3101, it can be difficult to write terminal independent code using READTEXT/PRINTEXT instructions. However, $IMAGE can be used to perform terminal independent input/output.

Static Screen Device Independence

Screen design for both the 4978/4979 and 3101 can be as simple as screen design for only the 4978/4979. This section describes how to design such terminal independent static screens, and discusses a limitation in compatibility between the 4978/4979 and 3101.

This section mentions both the $IMAGE utility and the $IMAGE subroutines. For a complete description of the $IMAGE utility, see the *Operator Commands and Utilities Reference*. For descriptions of the $IMAGE subroutines, see "$IMAGE Subroutines" on page PG-281 in this chapter.

Designing Terminal-Independent Static Screens

The $IMAGE utility and subroutines treat an unprotected field as a string of unprotected characters. In the 4978/4979 unprotected characters are null characters. If the $IMAGE null character were the at sign (@), then an unprotected field, eight characters long, could be defined as:

```
ENTER NAME HERE ==>  @@@@@@@@
```

This field could be defined the same way for a 3101; $IMAGE automatically inserts the attribute characters. In this case, the attribute byte immediately preceding the unprotected field would specify an unprotected and high intensity field. Somewhere preceding the protected field (ENTER NAME HERE) would be an attribute byte specifying a protected and low intensity field. Thus, if you do not want to define unique attributes (such as blinking), you can design screens for the 4978/4979 and use them on 3101 terminals with default attributes.

You can also design 3101 screens with unique attribute characters; in this case, a 3101 data stream is created by $IMAGE as well as a 4978/4979 image. The 3101 data stream is ignored for display on the 4978/4979. If the pound sign ('#') were defined as the blinking attribute, both fields in the previous example could be made to blink as follows:

```
#ENTER NAME HERE ==> #@@@@@@@@
```

On a 3101, a blinking, protected attribute byte would replace the first pound sign and a blinking, unprotected attribute byte would replace the second pound sign. The pound sign does not change the protect status of the field, merely its display properties; the "null" character determines whether the field is protected or unprotected.

Compatibility Limitation

This scheme has a limitation because an attribute byte is displayed as a protected blank. The character preceding a field (protected or unprotected) is always displayed as a blank on a 3101, even if a protected (non-blank) character appears on a 4978/4979. For example, the following screen is designed to display the month, day, and year as MM/DD/YY:

```
@@/@@/@@
```

On a 4978/4979, the date would appear as:

```
10/30/80
```

On a 3101, however, the date would appear as:

```
10 30 80
```

The slash characters on the 4978/4979 are replaced by attribute bytes on the 3101. Therefore, screens designed for the 4978/4979 do not have to be changed for use on the 3101. However, you have to alter them if you do not want protected characters to disappear when displayed on a 3101.

Coding EDL Instructions for Device Independence

To achieve static screen device independence between the 4978/4979 Display Station and the 3101 Display Terminal in block mode, you must use functionally equivalent terminal instructions on both terminals. The following considerations show one approach which provides some device independence.

- Use the 4978 screen images produced by $IMAGE for 4978/4979/3101 compatible applications. The 3101 data streams are not required.

- Specify an image type of C'4978' on calls to $IMOPEN.

- Specify FTAB on calls to $IMPROT. The FTAB buffer is initialized to describe each unprotected field on the screen and requires three words per entry.

- Use calls to $IMDATA to "scatter write" to either type terminal.

  PRINTEXT MODE=LINE does not produce a scatter write operation on the 3101 (as it does on the 4978/4979). A call to $IMDATA, specifying the FTAB produced by the prior call to $IMPROT and the user buffer, performs the scatter write operation on the 4978/4979 and simulates the scatter write on the 3101.

  $IMDATA can be used to write either default unprotected data from the screen image or user data contained in a user buffer.

- For "gather read" operations use:

  ```
  READTEXT MODE=LINE,TYPE=DATA,LINE=0,SPACES=0
  ```

  Read operations from the 3101 in block mode start with the first data field encountered, beginning with the upper left corner and continuing to the end of the screen. Specifying LINE=0,SPACES=0 makes the READTEXT from the 4978/4979 functionally equivalent to the 3101.

  In addition, the 3101 prefixes each field transmitted with three bytes of control information; this results in a 3101 data stream. Although EDX compresses out this control information, the user buffer must be large enough to contain the entire data stream that is transmitted.

- Using care, individual fields can be changed with:

  ```
  PRINTEXT MODE=LINE,LINE= ,SPACES=
  ```

  1. When directed to a 3101, the PRINTEXT instruction first writes an attribute byte, followed by the text data. The data field thus appears displaced one position to the right when compared to the result of a PRINTEXT directed to the 4978/4979. To suppress writing an attribute byte to the screen, use:

     ```
     TERMCTRL SET,ATTR=NO
     ```

     prior to the PRINTEXT(s). After the last PRINTEXT, code TERMCTRL SET,ATTR=YES. The 4978/4979 ignores these TERMCTRL instructions.

  2. Be careful to ensure that the data being sent to the 3101 does not extend beyond one data field; if it does, it will overlay and eliminate existing attribute characters. Once the screen attributes are changed, the FTAB no longer represents the screen and $IMDATA operations will produce undesired results.

3. Writing protected nulls to create additional unprotected 4978/4979 fields is not supported in 3101 block mode. Avoid this practice.

- Avoid the combination of "count" and TYPE=DATA in the ERASE instruction. On the 3101, the erase starts at the current cursor position and continues to the end of screen; the count operand is ignored.

- Avoid the combinations of TYPE=DATA,MODE=LINE and TYPE=DATA,MODE=FIELD in the ERASE instruction. Although these combinations work as anticipated on the 4978/4979, the 3101 forces the MODE= parameter to SCREEN.

- Avoid the combination of "count", TYPE=ALL and MODE=FIELD in the ERASE instruction. The 3101 forces MODE=FIELD to MODE=LINE. The operation terminates when the count reaches zero or the current line ends, whichever occurs first.

- To erase unprotected fields which do not end at end of line or end of screen, use one of the following techniques:

1. Use a PRINTEXT instruction with LINE and SPACES parameters to write blank characters to each individual field, being careful not to change or eliminate 3101 attribute bytes.

   Note: If the 3101 screen attributes are changed or eliminated, then the screen format will no longer match the FTAB and the data will not be directed to the correct locations on the 3101 screen. To re-establish the screen, call $IMPROT before calling $IMDATA.

2. Use READTEXT TYPE=DATA to read all unprotected data from the screen into a user buffer. Next, blank out (or change) the appropriate fields in the buffer. Then use the 'USER' buffer features of $IMDATA to rewrite the unprotected data.

Using the $IMAGE Subroutines for Device Independence

This section presents a way to write terminal-independent applications that use static screens. Using this method, the $IMAGE utility creates screen images and stores them on disk or diskette. Later, your application program can display and use the images by calling system-provided subroutines. Collectively these subroutines are called the "$IMAGE subroutines".

There are seven $IMAGE subroutines; see "$IMAGE Subroutines" on page PG-281 for individual descriptions of each. Ordinarily, your programs will not need to use all seven.

The Basic Steps

This section describes the basic steps in an application program which displays and processes a static screen (with a size of 24 lines and 80 characters per line):

- Retrieve the screen
- Display the protected data
- Display and retrieve the unprotected data

Retrieving the Screen Format: The first step is to retrieve the screen format by calling $IMOPEN. The type operand specifies the type of format to be retrieved. If the type operand is set to blanks, the format retrieved corresponds to the type of terminal upon which the program is running. If a 3101 format is needed but unavailable, the 4978/4979 format is retrieved and converted dynamically to a 3101 data stream. For example:

```
         CALL     $IMOPEN,(DSNAME),(FORMAT),(TERMTYPE)

DSNAME   TEXT     LENGTH=15     format dataset name
FORMAT   BUFFER   n,BYTES       format buffer
TERMTYPE DATA     CL4'    '     adapt to running terminal
```

Displaying the Protected Data: The screen format itself (the protected data) can be displayed with a call to $IMPROT.

```
         CALL     $IMPROT,(FORMAT),(FTAB)

FTAB     BUFFER   n,WORDS       field table
```

For the 3101, the field table (FTAB) is required. For a description of the field table, see "$IMPROT Subroutine" on page PG-285.

Displaying the Unprotected Data: At this point many applications generate and then display some data in the unprotected fields. On a 4978/4979 you can use PRINTEXT MODE=LINE to perform a scatter write operation. However, since this is not supported on a 3101, you should use $IMDATA to perform the scatter write operation and thus preserve device independence.

$IMDATA writes all the unprotected fields in a screen image. When directing data to the 3101, the field table generated by $IMPROT must be used. To write default unprotected data, use the buffer containing the screen image or specify a user buffer containing the application-provided data.

When $IMDATA is used with a user buffer, the application program must:

- Set the characters 'USER' in the first four positions of the buffer
- Set the message length, excluding 'USER', in the buffer index word (buffer-4)

```
          MOVE      USERDATA,CUSER,DWORD set up user message
          MOVE      DATALEN,8            set message length
          MOVE      USERDATA+4,MESSAGE,(8,BYTES) get message
          CALL      $IMDATA,(USERDATA),(FTAB)
          •
          •
          •
USERDATA  BUFFER    12,BYTES,INDEX=DATALEN  for user data
MESSAGE   DATA      CL8'HI THERE'           data
CUSER     DATA      CL4'USER'
```

Retrieving the Unprotected Data: After the operator has entered data, all the data in the unprotected fields can be read by a single statement. Both the 4978/4979 and 3101 support a "gather read" using READTEXT MODE=LINE.

```
         READTEXT SCRNDATA,MODE=LINE
         .
         .
SCRNDATA BUFFER   n,BYTES
```

In this example, *n* is the number of data bytes being read plus three bytes per field being read.

A READTEXT with MODE=LINE into a buffer from a 3101 screen has some special considerations. A READTEXT to the 3101 always reads from the beginning of the screen, regardless of the cursor position specified by LINE and SPACES. The 3101 has only three read options: read the entire screen (TYPE=ALL), read all the unprotected fields (TYPE=DATA), or read only the modified unprotected data (TYPE=MODDATA). (For more information on 3101 read options, see "Reading Modified Data on the 3101" on page PG-292).

The data will be read concatenated into the buffer. But the buffer must be large enough to accommodate the data plus three bytes (TYPE=DATA and TYPE=ALL) or four bytes (TYPE=MODDATA) per unprotected field. This extra data includes escape sequences and attribute bytes which are edited out of the buffer before presentation to the application program (as long as the default of STREAM=NO is in effect).

Although the 4978 has the capability to read a specific unprotected field, the 3101 does not. To perform a similar operation, the application can read all the unprotected data and then use the field table lengths to displace into the buffer and arrive at the desired data field.

Using TERMCTRL SET,ATTR=NO

Both the 4978 and 3101 can do a PRINTEXT with LINE and SPACES to a specific screen coordinate. However, for the 3101, doing this has ramifications for subsequent I/O to the screen. When a PRINTEXT is issued to a 3101 without a previous TERMCTRL SET,ATTR=NO, the terminal support inserts an attribute byte. This attribute byte appears as a protected blank at the screen coordinate specified by LINE and SPACES, and the data follows. Normally, this displaces the data one byte to the right, and therefore the data writes over the next attribute byte (which usually describes a protected field).

For example, assume the screen coordinate 5,5 (LINE=5,SPACES=5) contains a ten byte unprotected field which the application wants to fill with ten Xs. If a PRINTEXT LINE=5,SPACES=5 of ten Xs is issued with no previous TERMCTRL SET,ATTR=NO, then an attribute byte is added and written at location 5,5 and the tenth X overwrites the next attribute byte for the following protected field. This leaves the screen with one large unprotected field instead of a 10 byte unprotected field followed by a protected field.

A subsequent READTEXT of the unprotected data will result in much more data being returned to the application than expected. In addition, the returned data stream might contain escape sequences and attribute bytes which on a subsequent PRINTEXT from the same buffer will cause the cursor to act unpredictably. The data will also be written incorrectly on the screen.

To avoid such problems, a TERMCTRL SET,ATTR=NO should always be issued before a PRINTEXT with LINE and SPACES. A TERMCTRL SET,ATTR=YES should follow the PRINTEXT.

Converting 4978 Screens for Use on the 3101

Many 4978-based applications can be converted to run on the 3101. In some cases, it is sufficient to convert uses of PRINTEXT MODE=LINE to calls to $IMDATA. If the application uses READTEXT to specify screen coordinates with LINE and SPACES, the technique described above in "Using TERMCTRL SET,ATTR=NO" can be used.

Screens might also need to be changed because the attribute bytes are displayed as protected blanks on the 3101; see "3101 Screen Formats" on page PG-274.

$IMAGE Subroutines

Formatted screen images can be created and saved in disk or diskette data sets using the $IMAGE utility. The $IMAGE subroutines can be used to retrieve and display these images. These subroutines provide support for both the 4978/4979 and 3101 in block mode. In addition, screen images created on a 4978/4979 can be presented on a 3101 and vice versa with use of these subroutines. The intermixing of terminal screen images is also described in the *Operator Commands and Utilities Reference.*

The $IMAGE subroutines perform screen formatting and input/output operations independent of the type of terminal upon which the application runs. The orientation is towards writing/reading all unprotected fields with one operation. In this context the data in unprotected fields is of primary concern.

Static screen applications use the $IMOPEN, $IMDTYPE, $UNPACK, $IMGEN, $IMGEN31, $IMGEN49, and $IMGEN3X subroutine packages to process static screens defined using the $IMAGE utility.

$IMAGE Subroutines *(continued)*

$IMDTYPE is required for all static screen applications. In addition, the $IMOPEN and $UNPACK subroutines are also required, plus one of the following:

- $IMGEN to intermix both 3101 and 4978 images, and to display those images on either device
- $IMGEN3X to intermix both 3101 and 4978 images, and to display those images on a 3101
- $IMGEN31 for 3101 images, and to display those images on a 3101
- $IMGEN49 for 4978 images, and to display those images on a 4978 or 4979

During link-edit the $IMxxxx subroutines are included with your application through the use of the autocall library. Normally $IMGEN is included. If you want one of the alternate ($IMGENxx) routines, explicitly INCLUDE that module.

For formatted screen images presented on a 3101, storage requirements and internal conversion time is reduced when you select only the subroutine support that processes 3101 images.

An EXTRN statement must be coded for each subroutine name that your program references. You must link-edit the subroutines with your application program. $AUTO,ASMLIB should be specified as the autocall library to automatically include the screen formatting subroutines. See Chapter 5, "Preparing an Object Module for Execution"for details on the AUTOCALL feature of $EDXLINK.

The CALL syntax for the subroutines should be coded exactly as shown. Where an address argument is required by the subroutine, the label of the variable enclosed in parentheses causes the address to the passed (see the CALL instruction in the *Language Reference*).

If an error occurs, the terminal I/O return code will be in the first word of the task control block (TCB). These errors can come from instructions such as PRINTEXT, READTEXT, and TERMCTRL.

$IMOPEN Subroutine

The $IMOPEN subroutine reads the designated image from disk or diskette into your program buffer. You can also perform this operation by using the DSOPEN subroutine or defining the data set at program load time, and issuing the disk READ instruction. Refer to the section "Screen Image Buffer Sizes" on page PG-288 to determine the size of the buffer. $IMOPEN updates the index word of the buffer with the number of actual bytes read. To access it code buffer-4.

```
label        CALL      $IMOPEN,(dsname),(buffer),(type),
                       P2=,P3=,P4=

Required:    dsname,buffer
Defaults:    type=C'4978'
Indexable:   None
```

Operands	***Description***
dsname	The label of a TEXT statement which contains the name of the screen image data set. A volume label can be included, separated from the data set name by a comma.
buffer	The label of a BUFFER statement allocating the storage into which the image data will be read. The storage should be allocated in bytes, as follows: `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. 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 whose format corresponds with the type of terminal enqueued. If neither a 4978/4979 or 3101 is enqueued (ENQT), a 4978 image format is assumed.
Px	Parameter naming operands. See the CALL instruction and chapter 1 in the *Language Reference*.

The following is an example of $IMOPEN:

```
         CALL     $IMOPEN,(IMGDS),(IMGBUFF),(IMGTYP)
         .
         .
         .
IMGDS    TEXT     'IMGDS,MYVOL'
IMGBUFF  BUFFER   1024,BYTES
IMGTYP   DATA     C'3101'
```

$IMAGE Subroutines *(continued)*

$IMOPEN Return Codes

The following are the return codes (returned in taskname+2) from the $IMOPEN subroutine.

Code	Condition
-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

$IMDEFN Subroutine

The $IMDEFN subroutine is used to construct an IOCB for a formatted screen image. The IOCB can also be coded 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 an IOCB for the static screen has been defined, 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 unprotected fields.

```
label          CALL      $IMDEFN,(iocb),(buffer),topm,leftm,
                         P2=,P3=,P4=,P5=

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

Operands ***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   terminal,SCREEN=STATIC
```

buffer The label of an area containing the screen image in disk storage format. The format is described in the section "Screen Image Buffer Sizes" on page PG-288.

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 the CALL instruction and Chapter 1 in the *Language Reference*.

The following is an example of $IMDEFN:

```
         ENQT     IMGIOCB
          .
          .
          .
         CALL     $IMDEFN,(IMGIOCB),(IMGBUFF),0,0
          .
          .
          .
IMGIOCB  IOCB     SCREEN=STATIC
IMGBUFF  BUFFER   1024,BYTES
```

$IMPROT Subroutine

This subroutine uses an image created by the $IMAGE utility to prepare the defined protected and blank unprotected 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 inherently 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.

```
label          CALL       $IMPROT,(buffer),(ftab),P2=,P3=

Required:      buffer,ftab (see note)
Defaults:      None
Indexable:     None
```

buffer The label of an area containing the screen image in disk storage format. The format is described in the section "Screen Image Buffer Sizes" on page PG-288.

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 the CALL instruction and Chapter 1 in the *Language Reference*.

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 will be used to limit the amount of field information stored, and the buffer index word at buffer-4 will be 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.

The following is an example of $IMPROT:

```
          CALL      $IMPROT,(IMGBUFF),(FTAB)
          PRINTEXT  FTAB,SPACES=FTAB+2           POSITION CURSOR
          READTEXT  INPUT,FTAB+3                 OPERATOR INPUT
            .
            .
            .
IMGBUFF   BUFFER    1024,BYTES
FTAB      BUFFER    3,WORDS
INPUT     TEXT      LENGTH=20
```

$IMPROT Return Codes

The following are the return codes (returned in taskname+2) from the $IMPROT subroutine.

Code	Condition
-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

$IMDATA Subroutine

$IMDATA can be called to display the initial data values for an image which is in disk storage format. $IMDATA is used:

- To display the unprotected data associated with a screen image, if the content of the buffer is a screen format retrieved via $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'.

A user buffer can be specified 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.

Required:	buffer,ftab (see note)
Defaults:	None
Indexable:	None

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 in block mode or if a user buffer is used in $IMDATA.

Px Parameter naming operands. See the CALL instruction and Chapter 1 in the *Language Reference*.

The following is an example of $IMDATA:

```
         CALL      $IMDATA,(IMGBUFF),(FTAB)
         PRINTEXT  FTAB,LINE=FTAB,SPACES=FTAB+2    POSITION CURSOR
           .
           .
           .
IMGBUFF  BUFFER    1024,BYTES
FTAB     BUFFER    300,WORDS
```

$IMDATA Return Codes

The following are the return codes returned (returned in taskname+2) from the $IMDATA subroutine:

Code	Condition
-1	Successful completion
9	Invalid format in buffer

Screen Image Buffer Sizes

Under normal circumstances the size of the disk buffer can vary between 256 and 3096 bytes. Because data compression is used in storing the images, many images will require only 512 bytes, and 1024 bytes will be adequate for typical applications using 4978/4979 images. 3101 data stream images are much larger.

The $IMAGE utility tells you the required buffer sizes for the 4978 and 3101 buffers. If your application program will run on either type of terminal, use the larger of the two buffer sizes.

The display subroutines normally write images to the terminal in line-by-line fashion. Performance can be improved by providing a terminal buffer large enough to contain multiple lines. Since the display subroutines perform concatenated write operations whenever possible, using a larger buffer results in fewer such operations and, therefore, faster generation of the display image.

For example, for a full screen image (24 x 80), a time vs. space trade-off can be made by choosing a buffer size that is a multiple of 80 bytes (1 line), up to a maximum of 1920 bytes. A temporary buffer can be defined by coding the BUFFER= parameter on the IOCB which is used to access the screen. This buffer should be unique and should not be confused with the disk image buffer.

Example of Using $IMAGE Subroutines

The following program shows the $IMAGE subroutines in a general application program. Under direction of the terminal operator, this program displays on a 4978, 4979 or 3101 any image stored on disk. For each image, a field table (ftab) is constructed and used to modify initial data values.

In this example, use of the field size from the field table is for illustrative purposes only. Each unprotected output operation is terminated by the beginning of the next protected field, unless MODE=LINE is coded.

Additional examples on the use of the $IMAGE subroutines are in the appendix of the *Language Reference*.

```
IMDISP   PROGRAM   BEGIN
         EXTRN     $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
*
*           GET TERMINAL NAME FOR SCREEN PRINTOUT
*
BEGIN    READTEXT  IMAGE,'TERMINAL: '
*
*           GET IMAGE DATA SET NAME
*
         READTEXT  DSNAME,'DATA SET: ',PROMPT=COND
*
*           OPEN IMAGE DATA SET
*
         CALL      $IMOPEN,(DSNAME),(DISKBFR)
         MOVE      CODE,IMDISP+2     * SAVE RETURN CODE
         IF        CODE,NE,-1        * CHECK RETURN CODE FOR ERRORS
           PRINTEXT  '@OPEN ERROR CODE'
           PRINTNUM  CODE            * PRINT ERROR CODE
           GOTO      NEXT            * ASK IF TRY AGAIN
         ENDIF
*
*        CONSTRUCT IOCB
*
         CALL      $IMDEFN,(IMAGE),(DISKBFR),0,0
         ENQT      IMAGE            *  ACQUIRE STATIC SCREEN
         TERMCTRL  BLANK            *  BLANK SCREEN
*                                   *  WRITE PROTECTED FIELDS
*                                   *  AND BUILD FIELD TABLE
*                                   *  AT FTAB
```

```
*        DISPLAY PROTECTED FIELD DATA ON
*        TERMINAL SCREEN
*
         CALL      $IMPROT,(DISKBFR),(FTAB)
*
*        DISPLAY DEFAULT DATA ON
*        TERMINAL SCREEN
*
         CALL      $IMDATA,(DISKBFR),(FTAB)
*                                * SET CURSOR AT 1ST FIELD
         PRINTEXT  LINE=FTAB,SPACES=FTAB+2
         TERMCTRL  DISPLAY       * UNBLANK SCREEN
         DEQT                    * RETURN TO THIS TERMINAL
         WAIT      KEY           * WAIT FOR OPERATOR
         ENQT      IMAGE         * BACK TO TARGET TERMINAL
         TERMCTRL  BLANK         * BLANK SCREEN
*
*        DISPLAY #'S IN DATA FIELDS
*
         ENQT      IMAGE         * ACQUIRE STATIC SCREEN
         CALL      $IMDATA,(REPBFR),(FTAB)
         DEQT
         WAIT      KEY           * ALLOW VIEWING TIME
         ENQT      IMAGE         * ACQUIRE STATIC SCREEN
         ERASE     LINE=0,MODE=SCREEN,TYPE=ALL  * ERASE
         DEQT                    * BACK TO ROLL SCREEN
NEXT     QUESTION  'ANOTHER IMAGE? ',YES=BEGIN
         PROGSTOP
DSNAME   TEXT      LENGTH=16      * DATA SET NAME
*
*        BUILD A BUFFER OF #'S FOR A SECOND DATA
*        FIELD DISPLAY
*
B1       DC        F'72'          * B1 AND B2 INDEX REPBFR
B2       DC        F'76'          * THAT HIGHLIGHTS THE DATA
REPBFR   DC        C'USER'        * FIELDS FOR USER
         DC        C'####################################'
         DC        C'####################################'
DISKBFR  BUFFER    1064,BYTES     * DISK BUFFER
         DC        X'0808'        * TEXT CONTROL FOR NAME
IMAGE    IOCB      SCREEN=STATIC  * IOCB FOR IMAGE
CODE     DC        F'0'           * RETURN CODE
FTAB     BUFFER    300
LINE     TEXT      LENGTH=80
         ENDPROG
         END
```

Reading Modified Data

Reading modified data is supported on the 4978 Display Station and the 3101 Display Terminal; it is not supported on the 4979.

Reading Modified Data on the 4978

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

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

Reading Modified Data on the 3101

On the 3101, an unprotected field is considered to be a modified field when:

- Any character within the field is changed by the operator
- Certain ERASE instructions are executed
- The modified data tag (MDT) in the attribute byte is on

The modified data tags are reset when the data is read by a READTEXT TYPE=MODDATA instruction or transmitted by pressing the SEND key. To return a protected field using READTEXT TYPE=MODDATA, design the field with the modified data tag set on in the attribute byte.

To read all the modified fields from a screen, the operator must position the cursor on a protected line which does not contain any modified fields. If the cursor is not on such a line and the operator presses the enter key to satisfy a WAIT KEY instruction, the MDTs on that line are reset. A subsequent READTEXT would therefore not return to the program the modified data on that line. If a PF key instead of the SEND key is used to satisfy the WAIT KEY, the MDTs are not changed.

The IOCB BUFFER= parameter or the CCB buffer must be large enough to contain the received 3101 data stream prior to editing of the ESC sequences (four bytes for each modified field). If the CCB buffer is not large enough, use the IOCB buffer.

$UNPACK and $PACK Subroutines

The $UNPACK and $PACK subroutines move and translate compressed/noncompressed byte strings. These subroutines are used internally by the $IMPROT and $IMDATA subroutines as well as by the $IMAGE utility. However, they can also be called directly by an application program.

The program preparation needed for applications calling $UNPACK and $PACK is similar to that needed for the $IMAGE subroutines. An EXTRN statement is required in the application and the autocall to $AUTO,ASMLIB is required in the link-control data set (input to $EDXLINK).

$UNPACK Subroutine

This subroutine moves a series of compressed and noncompressed byte strings and translates the byte strings to noncompressed form.

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

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

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.

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

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

$PACK Subroutine

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

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

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

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.

4978/4979 Static Screen Sample Program

Line-oriented input/output instructions provide a straightforward way to construct and read data from static screens. However, when individual data fields on the 4978/4979 are accessed frequently, excessive screen flicker can result. This problem can be eliminated by transferring an entire screen image to the display with one I/O operation. Figure 12 shows this technique.

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

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

```
DISPLAY  PROGRAM   BEGIN
SCREEN   IOCB      SCREEN=STATIC,BOTM=11,                          C
              BUFFER=BUFF,RIGHTM=959
I        DATA      F'0'
BUFF     BUFFER    960,BYTES
         DATA      X'0202'
NULLS    DATA      X'0000'
NUMS     DATA      48F'0'
VALS     TEXT      LENGTH=254
BEGIN    ENQT      SCREEN
         ERASE     TYPE=ALL,LINE=0
         DO        96,INDEX=I
         PRINTEXT  'FIELD',PROTECT=YES
         PUTEDIT   FORMAT1,VALS,((I)),PROTECT=YES
         PRINTEXT  ' ',PROTECT=YES
         PRINTEXT  NULLS,PROTECT=YES
         ENDDO
         PRINTEXT  LINE=0
WRITE    PUTEDIT   FORMAT1,VALS,((NUMS,48)),                       C
              ACTION=STG
         PRINTEXT  VALS,MODE=LINE,LINE=0
         PRINTEXT  LINE=6,SPACES=8
         TERMCTRL  DISPLAY
         WAIT      KEY
         GOTO      (TRANSFER,QUIT),DISPLAY+2
```

Figure 12 (Part 1 of 2). 4978/4979 Static Screen Sample Program

4978/4979 Static Screen Sample Program *(continued)*

```
26 TRANSFER READTEXT  VALS,MODE=LINE,LINE=6
27          GETEDIT   FORMAT1,VALS,((NUMS,48)),                    C
                 ACTION=STG
29          ERASE     LINE=6,MODE=SCREEN,TYPE=DATA
30          GOTO      WRITE
31 QUIT     DEQT
            PROGSTOP
   FORMAT1  FORMAT (I2)
            ENDPROG
            END
```

Figure 12 (Part 2 of 2). 4978/4979 Static Screen Sample Program

The following numbers refer to lines (in the left margin) of the preceding figure:

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

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

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

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

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

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

13 Write ‘FIELD’ to the buffer.

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

15 Write a protected separation character.

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

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

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

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

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

23 Display the cursor at its defined position.

24 Wait for the operator to press an interrupt key.

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

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

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

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

30 Repeat.

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

3101 Static Screens

Summary of Design Considerations

The following list summarizes the items you should consider when designing a static screen application for the 3101.

- The 3101 uses a data stream which is a collection of special characters, commands, and data which tell the 3101 to do something.
- A simple PRINTEXT of 'HI THERE' results in a data stream of:

3101 Static Screens *(continued)*

```
ESC.Y.ROW.COL.ESC.3.ATTR.HI THERE
```

where ESC.Y is a set cursor address command followed by row and column position, and ESC.3 is a start-of-field followed by an attribute byte defining the field.

- An attribute byte defines how data will appear on the screen. It occupies one character position on the screen, and appears as a protected blank.
- Special attributes supported by the 3101 are high intensity, low intensity, blinking, and nondisplay.
- TERMCTRL SET,ATTR= sets the attribute byte.
- If an attribute is not required, a TERMCTRL SET,ATTR=NO should be done before a PRINTEXT to a specific X,Y location.
- Escape sequences take up space in the buffer. Therefore, it takes more than 1920 bytes to read a complete screen. Depending on the TERMCTRL SET ATTR= and TREAM= parameters in effect, a PRINTEXT operation could require the data length plus (7 x # fields). A READTEXT requires the data length plus (3 x # fields) for TYPE=ALL and TYPE=DATA, and the data length plus (4 x # fields) for TYPE=MODDATA.
- A READTEXT TYPE=DATA reads all unprotected data. If MODE=WORD, fields are separated by blanks. If MODE=LINE, fields are concatenated.
- A WAIT KEY prior to a READTEXT TYPE=MODDATA should be satisfied with a PF key and not the SEND key. If MODE=WORD, fields are separated by blanks. If MODE=LINE, fields are concatenated.
- A READTEXT without a prompt transmits data from the beginning of the screen, regardless of the cursor position.
- After the SEND key is pressed, a RDCURSOR returns as the cursor position the first position of the next line. If a PF key is pressed, it does not move the cursor.

3101 Static Screen Sample Program

This 3101 sample program shows the use of READTEXT/PRINTEXT coding for a static screen application. It is presented in three parts:

- "Coding Techniques" contains coding segments from the sample program with text explaining the functions and techniques used.
- "3101 Sample Program" shows the complete program. It can be executed on a 3101 in block mode.
- "Sample Program Output" shows the fields that are created on the display at program execution time.

EDX support for static screens on the 3101 is designed to read and write data by positioning the cursor, writing attribute bytes and issuing chained read buffer commands. These functions are sufficient to develop many formatted screen applications.

This sample program uses these functions and some additional 3101 hardware functions through the READTEXT, PRINTEXT, GETVALUE, RDCURSOR, GETEDIT, PUTEDIT, and PRINTNUM instructions.

A user buffer is not required unless a data stream read or written exceeds the CCB buffer size (generated from the TERMINAL statement). The techniques used perform input/output on a selected field-by-field basis. Therefore the data stream length is controlled so that it does not exceed the CCB buffer size (203 bytes for 3101 in block mode).

STREAM=YES is not used; this means that EDX will write the attribute bytes and escape sequences. The only exceptions are functions that EDX does not support; in these cases EBCDIC streams are used and EDX translates them.

In most cases, the attribute byte position is used for the LINE and SPACES parameters. This allows use of the same LINE/SPACES consistently for a given field; therefore, a field table approach could be used but is not shown in this example.

Coding Techniques

Enqueue the Static Screen

The program must enqueue to change the function to static screen. The screen size is forced to 24 x 80 and the CCB buffer of 203 bytes is used.

```
IOCB1     IOCB      SCREEN=STATIC
          .
          .
          .
ENQT      IOCB1
```

Change Attribute to Low Intensity

The default attribute is high intensity. After it is changed, this program always restores it to high intensity.

```
TERMCTRL SET,ATTR=LOW
```

Erase the Entire Screen

Erasing the screen defaults the count to 1920.

```
ERASE    TYPE=ALL
```

Protect the First Field

The first field defined is a protected field at 0,0. This ensures that the whole screen will be formatted and that no unformatted data areas will be returned on whole screen reads, whether the read is TYPE=ALL, TYPE=DATA or TYPE=MODDATA.

Printing the null character (defined in the DATA statement ATTRBUTE) with STREAM=NO in effect to LINE/SPACES causes EDX to:

- Generate the set cursor address sequence to the LINE/SPACES specified
- Generate the start field sequence, including the current attribute which will create or cause an attribute at LINE/SPACES to be rewritten

The data stream is shown below; the attribute byte is shown as '#'.

```
ESC.Y.ROW.COL.ESC.3.#.X'00'
```

The null data is required to force the start field sequence; however, a null character is ignored by the 3101.

```
          DATA    X'0101'           DUMMY TEXT STATEMENT CNT=1 LGTH=1
ATTRBUTE  DATA    X'0000'           NULL TO FORCE ATTRIBUTE TO WRITE
          .
          .
          .
          PRINTEXT ATTRBUTE,LINE=0,SPACES=0,PROTECT=YES
          TERMCTRL SET,ATTR=HIGH      RESTORE ATTRIBUTE
```

Create Unprotected Fields

To create unprotected fields on the screen ("holes" in which the operator can enter data), start each field with an unprotected attribute byte and end it with a protected attribute byte.

```
PRINTEXT    ATTRBUTE,LINE=4,SPACES=29
.
.
.
TERMCTRL    SET,ATTR=LOW
PRINTEXT    ATTRBUTE,LINE=4,SPACES=34,PROTECT=YES
```

Write Protected Fields

The next step is to create protected field descriptions. This could be done with ATTR=NO since the screen is already defined as protected in these areas. This program, however, uses a standard PRINTEXT to write a protected attribute byte at LINE/SPACES, followed by the literal data.

```
         PRINTEXT HEAD1,LINE=1,SPACES=20,PROTECT=YES
         PRINTEXT 'ENTER A NUMBER',LINE=4,SPACES=2,PROTECT=YES
```

Write a Nondisplay Field

The program uses a field description which is not initially displayed on the screen. To create a nondisplay field, set the attribute to blank.

```
NONDISP  TERMCTRL SET,ATTR=BLANK
         PRINTEXT 'ENTER ANOTHER NUMBER',LINE=12,SPACES=2,PROTECT=YES
         TERMCTRL SET,ATTR=HIGH     RESTORE ATTRIBUTE
```

GETVALUE using LINE/SPACES

Two EDL instructions that have an implied wait are:

- READTEXT with prompt
- GETVALUE with prompt

The LINE and SPACES parameters of these instructions specify the position of the attribute byte of the unprotected prompt field. Printing a null prompt field positions the attribute byte and cursor differently than for a prompt which is data. For example:

```
         Normal GETVALUE       = #prompt#_
         Null prompt GETVALUE = #_

NULPRMPT TEXT     LENGTH=0         USED ON IMPLIED WAIT INSTRUCTIONS
         .
         .
         .
GETVAL   GETVALUE FIELD1NO,NULPRMPT,LINE=4,SPACES=29
```

Write a Blinking Field

The program also uses a protected blinking field.

```
BLINK    TERMCTRL SET,ATTR=BLINK
         PRINTEXT 'FIELD1 MUST BE EVEN ',LINE=2,SPACES=5,PROTECT=YES
         TERMCTRL SET,ATTR=HIGH    RESTORE ATTRIBUTE
```

Erase Individual Fields

The program erases individual fields using the erase end-of-field/end-of-line function of the

3101 Static Screens *(continued)*

3101. To do this an ESC.I is sent as data. The field to be erased is specified by LINE/SPACES, and the current attribute byte is rewritten followed by the ESC.I. The data stream looks like:

```
          ESC.Y.ROW.COL.ESC.3.#.ESC.I


          DATA     X'0202'          ERASE END OF FIELD
ERASEFLD  DATA     X'27C9'          ESC.I
           .
           .
           .
ERASEF    PRINTEXT ERASEFLD,LINE=4,SPACES=29
```

Blank the Blinking Field

Once an even number is entered, the blinking field is blanked out by changing the attribute byte to nondisplay.

```
          TERMCTRL SET,ATTR=BLANK
          PRINTEXT ATTRBUTE,LINE=2,SPACES=5,PROTECT=YES
```

Simulate a Scatter Write

To simulate a scatter write, a horizontal tab character is inserted between fields. This is done using PUTEDIT; however, you could also use the CONCAT instruction or indexed moves. The data stream is shown below; an EBCDIC tab is a X'05'.

```
         ESC.Y.ROW.COL.ESC.3.#.DATA1.HT.DATA2

         DATA     X'0101'            HORIZONTAL TAB
TAB      DATA     X'0500'            TAB TO NEXT FIELD
          .
          .
          .
SCATTER  PUTEDIT FORMAT1,TEXTOUT,(AS,TAB,BS),LINE=6,SPACES=29
          .
          .
          .
FORMAT1  FORMAT   (A15,A1,A15),PUT
TEXTOUT  TEXT     LENGTH=31          SIZE OF DATA STREAM
```

Use the QUESTION Instruction

The program uses a standard QUESTION instruction.

```
QUEST    QUESTION 'WANT TO SEE MORE ?',NO=ENDIT,LINE=10,SPACES=5
```

An invalid response to a QUESTION (anything other than Y or N) is handled by the supervisor, which re-issues the read. This results in a string of two new fields: a question mark and a response field.

```
#PROMPT#?#?#?#?#_
```

Find and Blank Fields

To clear this string of fields, you could overwrite them with a protected field of blanks. Instead, this program finds each field and changes the attribute to blank protected.

```
         RDCURSOR LINE,SPACES       FIND CURSOR
         PRINTEXT LINE=LINE,SPACES=SPACES
         TERMCTRL DISPLAY           FORCE SOFT CURSOR ADDRESS
*                                   TO BE UPDATED
         DO       UNTIL,(SPACES,EQ,5),AND,(LINE,EQ,10)
```

A backtab command is sent as data to position the cursor in the first position of the unprotected field preceding the current cursor address. SET,ATTR=NO is used to prevent EDX from generating the attribute byte and preceding start field sequence. The data stream looks like:

```
         ESC.2


         DATA     X'0202'
BACKTAB  DATA     X'27F2'           BACK TAB TO FIRST CHARACTER
*                                   POSITION OF NON-PROTECTED FIELD
           .
           .
           .
           TERMCTRL SET,ATTR=NO
           PRINTEXT BACKTAB
           RDCURSOR LINE,SPACES     FIND NON-PROTECTED FIELD CURSOR
*                                   IS IN
           SUB      SPACES,1        ADJUST TO ATTRIBUTE BYTE
           TERMCTRL SET,ATTR=BLANK  PREPARE TO BLANK IT
           PRINTEXT ATTRBUTE,LINE=LINE,SPACES=SPACES,PROTECT=YES
        ENDDO
```

Change Field from Blank to Display

Now the program displays the nondisplay field previously discussed (ENTER ANOTHER NUMBER). The attribute that is currently blank protected is rewritten to low protected.

```
LIGHT    TERMCTRL SET,ATTR=LOW
         PRINTEXT ATTRBUTE,LINE=12,SPACES=2,PROTECT=YES
```

3101 Static Screens *(continued)*

Create a Data Entry Field

Next the program creates a new unprotected field with the cursor in place; this is useful for data entry. To create a unprotected field on demand with the cursor in place, write the end-of-field attribute first and then the start of field attribute.

```
CREATEU   TERMCTRL SET,ATTR=LOW
          PRINTEXT ATTRBUTE,LINE=12,SPACES=34,PROTECT=YES
          TERMCTRL SET,ATTR=HIGH      RESTORE ATTRIBUTE
          PRINTEXT ATTRBUTE,LINE=12,SPACES=29
          WAIT     KEY
```

Hardware Considerations for Reading Modified Data

A read of modified data has several implications:

- A field is modified by entering data or erasing the field. The modified data tag (MDT) in the attribute byte is turned on by the 3101.

- The modified data tag could be on when the attribute byte is written. $IMAGE provides this capability for 3101 data streams.

- Group 2, switch 4 on the 3101 enables the SEND key to function as the SEND LINE key. When the SEND key is pressed, the data that is on the same line as the cursor is sent. The type of data that is sent depends on the type of read in effect, namely all data, unprotected or modified.

- Once a modified field is sent to the Series/1 via the SEND key or a read buffer, the modified data tag in the attribute byte is turned off.

At this point during program execution, another number (FIELD4 data) has been entered and the SEND key has been pressed. The cursor was probably on the same line as FIELD4; if it was, FIELD4 data was sent to satisfy the WAIT KEY and the modified data tag was turned off. A subsequent READTEXT of TYPE=MODDATA would not return FIELD4 unless the cursor were moved to a line not containing modified fields, or a PF key were used to satisfy the WAIT KEY.

To read only the fields in which numbers were entered, the program re-writes the attribute bytes for those two fields with the modified data tags on. Before the modified fields are read, there is an intervening write, so the program locks the keyboard.

```
          TERMCTRL LOCK
          TERMCTRL SET,ATTR=NO         TO WRITE MDT ON ATTRIBUTE
```

Force Modified Data Tag On

A start field sequence with a unprotected, high intensity, MDT on attribute is written as data. The data stream looks like:

```
ESC.Y.ROW.COL.ESC.3.E
```

```
         DATA     X'0303'           TO FORCE MODIFIED DATA TAG ON
SETMOD   DATA     X'27F3'           START FIELD SEQUENCE
         DATA     X'C500'           ATTRIBUTE=HIGH,UNPROTECTED,MDT ON
           .
           .
           .
         PRINTEXT SETMOD,LINE=12,SPACES=29
         PRINTEXT SETMOD,LINE=4,SPACES=29
```

Read Modified Data

Now the program issues a READTEXT with TYPE=MODDATA; this reads all the modified data on the screen, in this case two fields.

```
READMOD  READTEXT MTEXT,TYPE=MODDATA,MODE=LINE
         IF       (MTEXT,NE,MTEXT+4,4)    PSEUDO TESTING
           .
           .
           .
MTEXT    TEXT     LENGTH=8          READ OF MODDATA:  STREAM
*                                   LENGTH = DATA + (4*NOFLDS) = 16
```

3101 Static Screens *(continued)*

Erase Fields Another Way

To erase a field, do an ERASE with a count value equal to the field length + 1 and TYPE=ALL. The + 1 is for the unprotected attribute.

```
ERASEF2  ERASE    5,TYPE=ALL,LINE=4,SPACES=29    ERASE FLD1
```

Erase to End of Screen

To prepare to erase the remaining fields, position the cursor to the second field.

```
         PRINTEXT LINE=6,SPACES=29
         TERMCTRL DISPLAY
```

Using ERASE with TYPE=DATA, all the unprotected fields from the current cursor position to the end of screen are erased. The count value is not used and mode is forced to screen.

```
ERASUNP  ERASE    TYPE=DATA          ERASE REMAINING UNPROTECT FIELDS
```

Read All Unprotected Fields

GETEDIT is used to get all the unprotected fields under format control. You could also use a READTEXT without a prompt; this would read all the unprotected data from the start of the screen.

```
GETALL    GETEDIT   FORMAT2,TEXTAMT,(NO1,ALPH1,ALPH2,NO2)
            .
            .
            .
TEXTAMT   TEXT      LENGTH=38           GETEDIT STREAM LENGTH =
*                                       DATA + (3*NOFLDS) = 50
FORMAT2   FORMAT    (I4,A15,A15,I4),GET
```

Write to LINE/SPACES

A standard PRINTNUM is used to write to LINE/SPACES.

```
          PRINTNUM NO1,FORMAT=(5,0,I),LINE=18,PROTECT=YES
```

Read from LINE/SPACES

To do a read from LINE/SPACES, a prompt field is required. The null prompt text statement (NULPRMPT) is used.

```
          TERMCTRL SET,ATTR=HIGH
          READTEXT TEXTIN,NULPRMPT,LINE=23,SPACES=70
```

3101 Sample Program

```
SAMPLE   PROGRAM   START
IOCB1    IOCB      SCREEN=STATIC
********************************************************************
*     EBCIDIC ESC SEQUENCES AND DATA STREAMS VIA PRINTEXT
********************************************************************
         DATA      X'0202'      ERASE END OF FIELD
ERASEFLD DATA      X'27C9'      ESC.I
         DATA      X'0303'      TO FORCE MODIFIED DATA TAG ON
SETMOD   DATA      X'27F3'      START FIELD SEQUENCE
         DATA      X'C500'      ATTRIBUTE=HIGH,UNPROTECTED,MDT ON
         DATA      X'0101'      DUMMY TEXT STATEMENT CNT=1,LGTH=1
ATTRBUTE DATA      X'0000'      NULL TO FORCE ATTRIBUTE TO WRITE
         DATA      X'0202'      ERASE END OF FIELD
BACKTAB  DATA      X'27F2'      BACK TAB TO FIRST CHARACTER
*                               POSITION OF NON-PROTECTED FIELD
         DATA      X'0101'      HORIZONTAL TAB
TAB      DATA      X'0500'      TAB TO NEXT FIELD
*********************************************************************
NULPRMPT TEXT      LENGTH=0          USED ON IMPLIED WAIT INSTUCTIONS
*********************************************************************
START    EQU       *
         ENQT      IOCB1
         TERMCTRL  SET,ATTR=LOW
         ERASE     TYPE=ALL
*                  START SCREEN WITH PROTECTED FIELD AT 0,0
         PRINTEXT  ATTRBUTE,LINE=0,SPACES=0,PROTECT=YES
         TERMCTRL  SET,ATTR=HIGH   RESTORE ATTRIBUTE
*                  CREATE NON PROTECTED FIELDS
         PRINTEXT  ATTRBUTE,LINE=4,SPACES=29
         PRINTEXT  ATTRBUTE,LINE=6,SPACES=29
         PRINTEXT  ATTRBUTE,LINE=8,SPACES=29
*                  NOW SET THE END OF NON PROTECTED FIELDS
         TERMCTRL  SET,ATTR=LOW
         PRINTEXT  ATTRBUTE,LINE=4,SPACES=34,PROTECT=YES
         PRINTEXT  ATTRBUTE,LINE=6,SPACES=45,PROTECT=YES
         PRINTEXT  ATTRBUTE,LINE=8,SPACES=45,PROTECT=YES
*                  CREATE PROTECTED LITERALS AS NEW FIELDS
*        THIS COULD BE DONE WITH ATTR=NO AS SCREEN IS PROTECTED
         PRINTEXT  HEAD1,LINE=1,SPACES=20,PROTECT=YES
         PRINTEXT  'ENTER A NUMBER',LINE=4,SPACES=2,PROTECT=YES
         PRINTEXT  'THIS IS FIELD2',LINE=6,SPACES=9,PROTECT=YES
         PRINTEXT  'THIS IS FIELD3',LINE=8,SPACES=9,PROTECT=YES
```

Figure 13 (Part 1 of 4). 3101 Static Screen Sample Program

```
*                       NON-DISPLAY THIS LITERAL FIELD AT THIS TIME
NONDISP  TERMCTRL  SET,ATTR=BLANK
         PRINTEXT  'ENTER ANOTHER NUMBER',LINE=12,SPACES=2        C
                   PROTECT=YES
         TERMCTRL  SET,ATTR=HIGH      RESTORE ATTRIBUTE
*                  NORMAL GETVALUE =    #PROMPT#_
*                  NULL PROMPT GETVALUE =    #_
GETVAL   GETVALUE  FIELD1NO,NULPRMPT,LINE=4,SPACES=29
         DIVIDE    FIELD1NO,2,RESULT=DUMMY
         IF        (SAMPLE,NE,0)
*          *       CREATE NEW PROTECTED BLINKING FIELD
BLINK      TERMCTRL  SET,ATTR=BLINK
           PRINTEXT  'FIELD1 MUST BE EVEN ',LINE=2,SPACES=5,      C
                     PROTECT=YES
           TERMCTRL  SET,ATTR=HIGH     RESTORE ATTRIBUTE
*          *         GOING TO ERASE AN INDIVIDUAL FIELD USING ERASE
ERASEF     PRINTEXT  ERASEFLD,LINE=4,SPACES=29
           GOTO      GETVAL
         ELSE
*          *         BLANK OUT BLINKING FIELD BY GOING NON-DISPLAY
           TERMCTRL  SET,ATTR=BLANK
           PRINTEXT  ATTRBUTE,LINE=2,SPACES=5,PROTECT=YES
           TERMCTRL  SET,ATTR=HIGH        RESTORE ATTRIBUTE
*          *         DO SCATTER WRITE BY INSERTING TAB CHARACTER
SCATTER    PUTEDIT FORMAT1,TEXTOUT,(AS,TAB,BS),LINE=6,SPACES=29
         ENDIF
*                  GOING TO DO STANDARD QUESTION
QUEST    QUESTION  'WANT TO SEE MORE ?',NO=ENDIT,LINE=10,SPACES=5
*        *         QUESTION AND INVALID RESPONSES CAN YIELD
*        *         #PROMPT#?#?#?#?#_
*        *         NEED TO FIND ALL ATTRIBUTES '#' AND CLEAR
         RDCURSOR    LINE,SPACES       FIND CURSOR
         PRINTEXT    LINE=LINE,SPACES=SPACES
         TERMCTRL    DISPLAY    FORCE SOFT CURSOR ADDRESS
*                               TO BE UPDATED
         DO        UNTIL,(SPACES,EQ,5),AND,(LINE,EQ,10)
           TERMCTRL  SET,ATTR=NO
           PRINTEXT  BACKTAB
           RDCURSOR  LINE,SPACES   FIND NON-PROTECTED FIELD CURSOR
*                                  IS IN
           SUB       SPACES,1      ADJUST TO ATTRIBUTE BYTE
           TERMCTRL  SET,ATTR=BLANK    PREPARE TO BLANK IT
           PRINTEXT  ATTRBUTE,LINE=LINE,SPACES=SPACES,PROTECT=YES
         ENDDO
*                  LIGHT UP NON DISPLAY FIELD4 PROMPT
LIGHT    TERMCTRL  SET,ATTR=LOW
         PRINTEXT  ATTRBUTE,LINE=12,SPACES=2,PROTECT=YES
         PRINTEXT  'ON A WAIT KEY NOW',LINE=13,SPACES=9,PROTECT=YES
```

Figure 13 (Part 2 of 4). 3101 Static Screen Sample Program

```
*           CREATE NEW NON-PROTECTED FIELD WITH CURSOR IN PLACE
CREATEU  TERMCTRL  SET,ATTR=LOW
         PRINTEXT  ATTRBUTE,LINE=12,SPACES=34,PROTECT=YES
         TERMCTRL  SET,ATTR=HIGH    RESTORE ATTRIBUTE
         PRINTEXT  ATTRBUTE,LINE=12,SPACES=29
         WAIT      KEY
********************************************************************
*        LOCK THE KEYBOARD, RE-WRITE ATTRIBUTES WITH MDT ON,
*        READ THESE TWO FIELDS WITH TYPE=MODDATA
********************************************************************
         TERMCTRL  LOCK
         TERMCTRL  SET,ATTR=NO    TO WRITE MDT ON ATTRIBUTE
         PRINTEXT  SETMOD,LINE=12,SPACES=29
         PRINTEXT  SETMOD,LINE=4,SPACES=29
         TERMCTRL  SET,ATTR=YES   RESTORE
READMOD  READTEXT  MTEXT,TYPE=MODDATA,MODE=LINE
         IF        (MTEXT,NE,MTEXT+4,4)     PSEUDO TESTING
           TERMCTRL  SET,ATTR=BLINK
           PRINTEXT  'FLD4 MUST = FLD1  ',LINE=13,SPACES=9,      C
                     PROTECT=YES
           TERMCTRL  SET,ATTR=HIGH       RESTORE
ERASEF2    ERASE     5,TYPE=ALL,LINE=4,SPACES=29   ERASE FLD1
           PRINTEXT  LINE=6,SPACES=29
           TERMCTRL  DISPLAY
ERASEUNP   ERASE     TYPE=DATA   ERASE REMAINING UNPROTECTED FIELDS
           TERMCTRL  UNLOCK
           GOTO      GETVAL
         ENDIF
         TERMCTRL  UNLOCK
GETALL   GETEDIT   FORMAT2,TEXTAMT,(NO1,ALPH1,ALPH2,NO2)
         TERMCTRL  SET,ATTR=BLINK
         PRINTEXT  'YOU ENTERED:',LINE=16,PROTECT=YES
         TERMCTRL  SET,ATTR=HIGH
         PRINTNUM  NO1,FORMAT=(5,0,I),LINE=18,PROTECT=YES
         PRINTEXT  ALPH1,LINE=19,PROTECT=YES
         PRINTEXT  ALPH2,LINE=20,PROTECT=YES
         PRINTNUM  NO2,FORMAT=(5,0,I),LINE=21,PROTECT=YES
         TERMCTRL  DISPLAY
ENDIT    EQU       *
         TERMCTRL  SET,ATTR=LOW
         PRINTEXT  'IF YOU WANT TO SEE IT AGAIN ENTER ''AGAIN'' >',  X
               LINE=23,SPACES=5,PROTECT=YES
         TERMCTRL  SET,ATTR=HIGH
```

Figure 13 (Part 3 of 4). 3101 Static Screen Sample Program

```
*                      FINALLY A READTEXT TO LINE AND SPACES
         READTEXT      TEXTIN,NULPRMPT,LINE=23,SPACES=70
         IF            (TEXTIN,EQ,CAGAIN,5),GOTO,START
         PROGSTOP      LOGMSG=NO
FORMAT1  FORMAT        (A15,A1,A15),PUT
TEXTOUT  TEXT          LENGTH=31          SIZE OF DATA STREAM
LINE     DATA          F'0'
SPACES   DATA          F'0'
FIELD1NO DATA          F'0'
HEAD1    TEXT          '*** 3101 SAMPLE PROGRAM ***'
MTEXT    TEXT          LENGTH=8   READ OF MODDATA LGTH=DATA + (4*NOFLDS)
DATABFR  DATA          C'AAAAAAAAAAAAAAABBBBBBBBBBBBBBB'
AS       EQU           DATABFR
BS       EQU           AS+15
TEXTAMT  TEXT          LENGTH=38  GETEDIT STREAM LGTH= DATA + (3*NOFLDS)
FORMAT2  FORMAT        (I4,A15,A15,I4),GET
NO1      DATA          F'0'
NO2      DATA          F'0'
ALPH1    TEXT          LENGTH=15
ALPH2    TEXT          LENGTH=15
CAGAIN   DATA          C'AGAIN'
TEXTIN   TEXT          ' ',LENGTH=5
DUMMY    DATA          F'0'
      ENDPROG
      END
```

Figure 13 (Part 4 of 4). 3101 Static Screen Sample Program

3101 Static Screens *(continued)*

Sample Program Output

Figure 14 shows the fields that the 3101 sample program creates on the display at program execution time.

```
                      !3101 SAMPLE PROGRAM
    $FIELD1 MUST BE EVEN

 !ENTER A NUMBER            @....!

       !THIS IS FIELD2      @..............!

       !THIS IS FIELD3      @..............!

    @..................@......................................................
..............................................................................
..%ENTER ANOTHER NUMBER     @....!
      !ON A WAIT KEY NOW


$YOU ENTERED:

@.............................................................................
@.............................................................................
@.............................................................................
@.............................................................................
..............................................................................
.....!IF YOU WANT TO SEE IT AGAIN ENTER 'AGAIN'>                     @........
```

Figure 14. 3101 Sample Program Output

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.

Glossary of Terms and Abbreviations

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.

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).

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 control 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 recreate data that has been lost or damaged.

reflective marker. A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative 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.

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

Special Characters

Index

C

D

P

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

IBM Series/1 Event Driven Executive
Language Programming Guide
SC34-0438-0

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.*

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.)

SC34-0438-0

Printed in U.S.A.

Reader's Comment Form

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

Cut or Fold Along Line

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

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

IBM®

Series/1

IBM

SC34-0438-0

Event Driven Executive Language Programming Guide

Version 4.0

Reference Cards

Event Driven Language Programming Guide

Operation Guide

Series/1

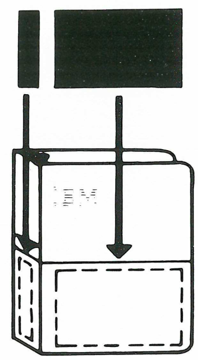
Event Driven Executive
Language
Programming Guide

PG

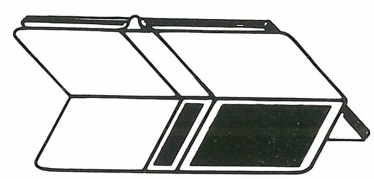
Binder Labels

Tear this page along the perforations to separate the two labels.

Insert the labels into the clear plastic sleeves.



To stand the easel binder up, open it and fold it as shown.



IBM®

International Business Machines Corporation

SC34-0438-0
Program Numbers: 5719-XS4, 5719-XX5
File No.: S1-20
Printed in U.S.A.