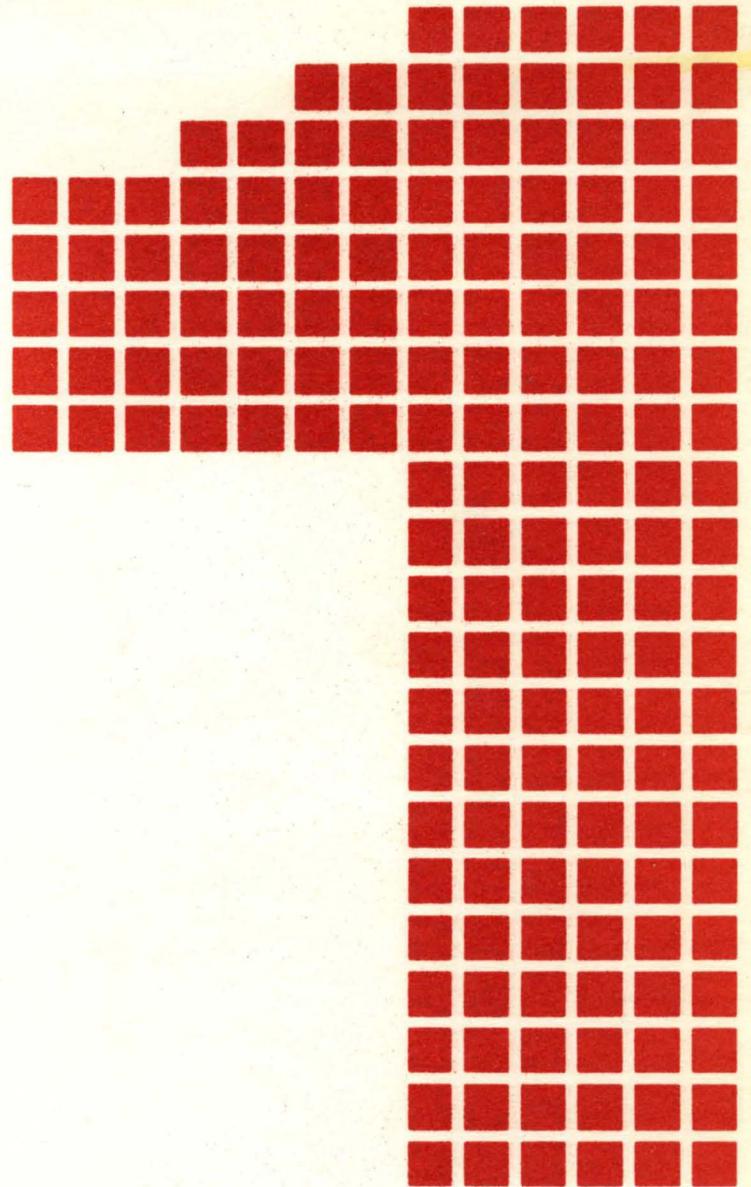




SR30-0436-1



IBM Series/1  
Event Driven Executive  
Licensed Program  
Study Guide



SR30-0436-1

**IBM Series/1  
Event Driven Executive  
Licensed Program  
Study Guide**

## **Second Edition (September 1980)**

This edition is a major revision of, and obsoletes, SR30-0436-0. It contains material in support of IBM Series/1 Event Driven Executive version 3.

Changes are periodically made to the information herein; any such changes will be reported in subsequent editions.

Use this publication only for the uses stated in "Section 1. Introduction to This Course."

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

This publication could contain technical inaccuracies or typographical errors.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, General Systems Division, Technical Publications, Department 796, P.O. Box 2150, Atlanta, Georgia 30055. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

# Contents

<b>Section 1. Introduction to This Course</b> . . . . .	1-1	<b>Section 5. Data Manipulation</b> . . . . .	5-1
Course Overview . . . . .	1-1	Integer Arithmetic . . . . .	5-1
Material Requirements . . . . .	1-3	Optional Operands . . . . .	5-2
Study Tips . . . . .	1-3	Floating Point Arithmetic . . . . .	5-3
Course Objectives . . . . .	1-4	Data Movement Instructions . . . . .	5-4
Event Driven Executive Components—Version 3 . . . . .	1-4	Logical Instructions . . . . .	5-6
Basic Supervisor and Emulator (5719-XS3) . . . . .	1-4	Data Manipulation Review Exercise — Questions . . . . .	5-10
Event Driven Executive Utilities (5719-UT5) . . . . .	1-5	Data Manipulation Review Exercise — Answers . . . . .	5-12
Event Driven Executive Macro Library/Host (5740-LM4) . . . . .	1-5	<b>Section 6. Queue Processing</b> . . . . .	6-1
Event Driven Executive Program Preparation Facility (5719-XX4) . . . . .	1-5	DEFINEQ . . . . .	6-2
Event Driven Executive Macro Library (5719-LM7) . . . . .	1-6	LASTQ/FIRSTQ/NEXTQ . . . . .	6-4
Event Driven Executive — An Operational Overview . . . . .	1-6	Queue Processing Review Exercise — Questions . . . . .	6-7
<b>Section 2. Instruction Format</b> . . . . .	2-1	Queue Processing Review Exercise — Answers . . . . .	6-8
Event Driven Executive Basic Instruction Format . . . . .	2-1	<b>Section 7. Program Control</b> . . . . .	7-1
Language Syntax/Coding Conventions . . . . .	2-1	Subroutines . . . . .	7-1
Instruction Format . . . . .	2-2	SUBROUT Statement . . . . .	7-1
Instruction Format Review Exercise — Questions . . . . .	2-5	CALL Statement . . . . .	7-2
Instruction Format Review Exercise — Answers . . . . .	2-6	Passing Subroutine Parameters . . . . .	7-2
<b>Section 3. Program/Tasks</b> . . . . .	3-1	USER Statement . . . . .	7-5
Program/Task Concepts and Structure . . . . .	3-1	Program Control Review Exercise — Questions . . . . .	7-9
Single Task Program . . . . .	3-2	Program Control Review Exercise — Answers . . . . .	7-10
Multiple Task Programs . . . . .	3-3	<b>Section 8. Program Sequencing</b> . . . . .	8-1
Multiple Program Structure . . . . .	3-5	GOTO Statement . . . . .	8-1
Overlay Program Structure . . . . .	3-7	IF Statement . . . . .	8-4
Program/Task Definition . . . . .	3-10	Relational Conjunctions . . . . .	8-6
Program/Task Execution . . . . .	3-12	DO Statement . . . . .	8-7
Program Loading . . . . .	3-12	Program Sequencing Review Exercise — Questions . . . . .	8-11
Program Synchronization . . . . .	3-15	Program Sequencing Review Exercise — Answers . . . . .	8-14
Task Synchronization . . . . .	3-18	<b>Section 9. Timers</b> . . . . .	9-1
Queueable Resources . . . . .	3-21	GETTIME Instruction . . . . .	9-1
WAIT/POST Operation . . . . .	3-24	INTIME Instruction . . . . .	9-2
Attention Lists . . . . .	3-26	STIMER Instruction . . . . .	9-3
Programs/Tasks Review Exercise — Questions . . . . .	3-29	Timing Functions — Coding Example . . . . .	9-4
Programs/Tasks Review Exercise — Answers . . . . .	3-32	Timers Review Exercise — Questions . . . . .	9-7
<b>Section 4. Data Definition</b> . . . . .	4-1	Timers Review Exercise — Answers . . . . .	9-8
DATA Statement . . . . .	4-1	<b>Section 10. Disk/Diskette I/O</b> . . . . .	10-1
BUFFER Statement . . . . .	4-6	Devices Supported — Diskette . . . . .	10-1
TEXT Statement . . . . .	4-8	Devices Supported — Disk . . . . .	10-1
Data Definition Review Exercise — Questions . . . . .	4-11	Devices Supported — Tape . . . . .	10-1
Data Definition Review Exercise — Answers . . . . .	4-12	Disk Volume Definition . . . . .	10-2
		Diskette Volume Definition . . . . .	10-3

Tape Volume Definition	10-3
Data Sets	10-3
Records	10-4
Record/Sector Relationship	10-4
PROGRAM Statement DS= Operand	10-4
READ/WRITE Statements – Disk/Diskette	10-6
READ/WRITE Statements – Tape	10-10
NOTE/POINT Statements	10-11
Disk/Diskette I/O Coding Examples	10-11
Tape I/O Examples	10-17
Load-Time Data Set Definition	10-17
Disk/Diskette I/O Review Exercise – Questions	10-23
Disk/Diskette I/O Review Exercise – Answers	10-28
<b>Section 11. Terminal I/O</b>	<b>11-1</b>
TERMINAL Statement	11-1
Roll Screens	11-2
NHIST= Operand	11-2
Static Screens	11-3
ENQT/DEQT Instructions	11-4
IOCB Statement	11-6
Data Representation	11-8
PRINTEXT Instruction	11-8
READTEXT Instruction	11-16
Operator Control of Program Execution	11-19
PF and Attention Key Handling	11-19
QUESTION Instruction	11-21
WAIT KEY Instruction	11-22
HARDCOPY PF Key	11-23
Static Screen Coding Example	11-24
ERASE Instruction	11-26
TERMCTRL Instruction	11-27
RDCURSOR Instruction	11-43
PRINTNUM/GETVALUE Instructions	11-43
PRINTIME/PRINDATE Instructions	11-52
Terminal I/O Review Exercise – Questions	11-53
Terminal I/O Review Exercise – Answers	11-56
<b>Section 12. Data Formatting</b>	<b>12-1</b>
Data Conversion	12-1
CONVTD Instruction	12-2
CONVTB Instruction	12-3
CONVTD/CONVTB Coding Examples	12-4
GETEDIT/PUTEDIT Introduction	12-7
PUTEDIT/GETEDIT Instructions	12-10
FORMAT Statement	12-11
Data Formatting Review Exercise – Questions	12-19
Data Formatting Review Exercise – Answers	12-20
<b>Section 13. Sensor I/O</b>	<b>13-1</b>
Sensor Based I/O	13-1
Digital Input/Output	13-4
Analog Input/Output	13-4
Event Driven Executive Sensor I/O Support	13-6

IODEF Statement	13-8
SBIO Statement	13-10
Sensor I/O Coding Examples	13-12
Sensor I/O Review Exercise – Questions	13-19
Sensor I/O Review Exercise – Answers	13-20

## Introduction to Sections 14 Through 18

<b>Section 14. Utility Programs</b>	<b>14-1</b>
Operator Commands	14-1
\$A	14-1
\$B	14-2
\$C	14-2
\$D and \$P	14-2
\$CP	14-2
\$E	14-3
\$S	14-3
\$T and \$W	14-3
\$VARYON and \$VARYOFF	14-4
Operator Command	14-4
System Utility Programs – Introduction	14-6
Data Management/Maintenance Utilities	14-7
\$DASDI	14-7
\$INITDSK	14-9
\$DISKUT1	14-11
\$DISKUT2	14-12
\$COPY	14-14
\$COPYUT1	14-15
\$MOVEVOL	14-17
\$COMPRES	14-17
\$TAPEUT1	14-18
Terminal I/O Utilities	14-19
\$TERMUT1	14-19
\$TERMUT2	14-21
\$TERMUT3	14-25
\$PFMAP	14-25
\$FONT	14-26
Miscellaneous Utilities	14-36
\$IMAGE	14-36
\$IOTEST	14-43
\$PREFIND	14-44
Program Preparation Utilities	14-46
\$EDIT1N	14-46
\$UPDATE	14-49
\$FSEEDIT	14-50
Other Utility Programs	14-80
BSC Utilities	14-80
\$BSCTRCE	14-80
\$BSCUT1	14-80
\$BSCUT2	14-80
Display Processor (Graphics) Utilities	14-81
\$DIUTIL	14-81
\$DICOMP	14-81
\$DIINTER	14-81

Host Program Preparation Utilities . . . . .	14-81
\$HCFUT1 . . . . .	14-82
\$EDIT1/\$UPDATEH . . . . .	14-82
\$RJE2780/\$RJE3780 . . . . .	14-82
\$PRT2780/\$PRT3780 . . . . .	14-83
\$DEBUG . . . . .	14-83
<b>Section 15. System Installation . . . . .</b>	<b>15-1</b>
Objectives . . . . .	15-1
Machine Readable Material . . . . .	15-1
5719-XS3 Basic Supervisor and Emulator . . . . .	15-1
5719-UT5 System Utility Programs . . . . .	15-1
5719-XX4 Program Preparation Facility . . . . .	15-1
5719-LM7 Macro Library . . . . .	15-2
5740-LM4 Macro Library/Host . . . . .	15-2
Starter System Installation Overview . . . . .	15-2
Installing the Starter System . . . . .	15-3
User System Generation . . . . .	15-5
SYSGEN Overview . . . . .	15-5
Allocate Required Data Sets . . . . .	15-5
Edit System Configuration Statements . . . . .	15-7
\$IOTEST . . . . .	15-7
SYSTEM Statement . . . . .	15-10
TIMER Statement . . . . .	15-11
DISK Statements . . . . .	15-11
TERMINAL Statement . . . . .	15-12
System Communications Area . . . . .	15-13
Select Supervisor Support Modules . . . . .	15-15
Edit \$JOBUTIL Procedure File . . . . .	15-19
Assemble/Link/Format . . . . .	15-20
Designate Tailored Supervisor . . . . .	15-24
IPL Tailored Supervisor . . . . .	15-26

<b>Section 16. Program Preparation Using</b>	
<b>\$EDXASM . . . . .</b>	<b>16-1</b>
Program Preparation Overview . . . . .	16-1
\$EDXASM . . . . .	16-3
\$EDXLIST . . . . .	16-5
\$LINK . . . . .	16-5
\$JOBUTIL . . . . .	16-7
Program Preparation Example . . . . .	16-12
Problem Description . . . . .	16-12
Create/Modify Source Module . . . . .	16-13
Assemble Source Module . . . . .	16-22
Produce Assembly Listing . . . . .	16-24
Link Edit Object Modules . . . . .	16-25
Format Object Module . . . . .	16-28
\$EDXASM Copy Code Function . . . . .	16-30
Job Stream Procedure . . . . .	16-37

<b>Section 17. Program Preparation Using</b>	
<b>\$S1ASM . . . . .</b>	<b>17-1</b>
Objectives . . . . .	17-1
\$S1ASM Machine Readable Material . . . . .	17-1
Installing \$S1ASM . . . . .	17-2
\$S1ASM Operation . . . . .	17-2
Data Set Requirements . . . . .	17-3
\$S1ASM/\$JOBUTIL Interface . . . . .	17-5

<b>Section 18. Session Manager . . . . .</b>	<b>18-1</b>
Objectives . . . . .	18-1
Session Manager Overview . . . . .	18-1
Session Manager Operating Concepts . . . . .	18-6
Definition of Terms . . . . .	18-6
Using the Session Manager . . . . .	18-10
Loading the Session Manager . . . . .	18-10
Data Set Allocation . . . . .	18-12

<b>Appendix A. SYSGEN Listings . . . . .</b>	<b>A-1</b>
--	------------

<b>Appendix B. Program Preparation Listings . . . . .</b>	<b>B-1</b>
---	------------

This page intentionally left blank.

## Section 1. Introduction to This Course

This course is intended to give Series/1 personnel a general knowledge of the concepts and theory incorporated in the Event Driven Executive system Version 3. Upon completion of this course, the student should be able to install, generate and maintain an Event Driven Executive system as well as write and execute basic application programs.

The prerequisite for this course is successful completion of *Introduction to Smaller Systems Student Text* (SR30-0185) or equivalent experience. Programming experience using high level languages is also strongly recommended.

### COURSE OVERVIEW

The Event Driven Executive instruction set and system support programs have been divided into several broad functional groups, each group constituting a section of this study guide. An attempt has been made to organize the sections in a logical sequence for study. Each section, however, is also as modular as possible, and can be studied as a separate unit, or in a sequence other than presented, if desired.

#### *Section 1. Introduction to This Course*

Contains introductory material, as well as a brief operational overview of the Event Driven Executive system.

#### *Section 2. Instruction Format*

Coding conventions/syntax rules for coding Event Driven Executive instructions.

#### *Section 3. Programs/Tasks*

This section covers program/task structure, application program design considerations, and all of the Event Driven Executive instructions used for task control and synchronization.

#### *Section 4. Data Definition*

#### *Section 5. Data Manipulation*

These two sections cover all of the basic instructions required to define, move, or perform logical or arithmetic operations on data in storage.

#### *Section 6. Queue Processing*

Discussion and illustration of the queue definition and processing instructions.

#### *Section 7. Program Control*

How to define and use both Event Driven Executive subroutines, and subroutines written in Series/1 Assembler Language.

***Section 8. Program Sequencing***

Discussion and illustration of IF and DO structures, and the relational statements used with them.

***Section 9. Timers***

Instructions to access the system's 24 hour clock and the elapsed time clock, and to wait for a time delay are discussed.

***Section 10. Disk/Diskette I/O***

Discussion and examples of defining and accessing data sets from an application program.

***Section 11. Terminal I/O***

***Section 12. Data Formatting***

The comprehensive terminal I/O support provided by the Event Driven Executive is discussed in detail, with several coding examples. Data Formatting support is used with terminals, and therefore immediately follows.

***Section 13. Sensor Input/Output***

This section includes some basic sensor I/O concepts, as well as how to incorporate the sensor I/O support in a supervisor and to access sensor I/O devices from a user program.

***Section 14. System Utilities***

All of the system utilities are described. Those utilities required most often are discussed in detail.

***Section 15. System Installation***

This section covers installation of the supplied supervisor and system programs as received from PID, and generation of a tailored supervisor, using the online Program Preparation Facility.

***Section 16. Program Preparation Using \$EDXASM***

\$FSEDIT (text editor), \$EDXASM (Event Driven language assembler), \$LINK (link editor), \$UPDATE (object module formatter), and \$JOBUTIL (job stream processor) are used to prepare a program for execution. Included are examples of the use of the COPY CODE assembler feature and the AUTOCALL link editor option.

***Section 17. Program Preparation Using \$S1ASM***

This optional topic is for those users who will be assembling Series/1 assembler language and/or Event Driven language programs using \$S1ASM, the Series/1 Macro Assembler (Licensed Program 5719-ASA).

***Section 18. Session Manager***

Organization and operation of the Session Manager programmer productivity tool.

## MATERIAL REQUIREMENTS

Course Materials	Form No.
* IBM Series/1 Event Driven Executive Licensed Program Study Guide	SR30-0436
<b>Additional Materials</b>	
* IBM Series/1 Event Driven Executive System Guide	SC34-1702
* IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes	SC34-1703
* IBM Series/1 Event Driven Language Reference	SC34-1706
** IBM Series/1 Macro Assembler Language Reference	SC34-0317
*** IBM Series/1 Event Driven Executive Communications and Terminal Application Guide	SC34-1705
*** IBM Series/1 Event Driven Executive Internal Design	LY34-0202
* Required to complete this course.	
** Not required to complete course, but may be a useful reference for users who will be preparing programs written in Series/1 assembler language, using the Series/1 Macro Assembler (5719-ASA) (optional topic in Study Guide).	
*** Not required to complete this course. These manuals address topics that are not covered in the study guide, but which may be of interest to some students.	

## STUDY TIPS

Each section has a set of objectives. Read the objectives carefully so that you understand what you should be learning in that section. For each topic you will find a **READING ASSIGNMENT**. Read the reading assignment and then continue in the Self Study Guide. At the end of most sections you will find a **Review Exercise**. Try to complete it prior to looking at the correct answers and be sure you understand your mistakes before proceeding to the next topic or section.

The total amount of study time you will need is estimated at 50 to 60 hours. This may extend over a period of two or three weeks if your study periods are brief and somewhat separated because of other duties.

For best results, set a short time goal rather than a long one and then make every effort to meet that goal. Study sessions should be about 2 hours long but use whatever time you wish. You may find that several short sessions are more productive than one longer session.

## **COURSE OBJECTIVES**

The student upon completion of this self-study course should be able to:

1. Describe the major components and facilities of the Series/1 Event Driven Executive system
2. Install an Event Driven Executive system on a Series/1
3. Use the utility programs to maintain a system
4. Invoke Supervisor utility functions from a terminal
5. Use most of the Event Driven Executive instructions necessary to code application programs
6. Load application programs from a terminal, or from other programs
7. Understand the use of overlay programs, multitasking, and task/program synchronization

## **EVENT DRIVEN EXECUTIVE COMPONENTS—VERSION 3**

The Event Driven Executive software offering consists of five licensed programs:

- |  |            |
|--|------------|
| 1. Basic Supervisor and Emulator                       | (5719-XS3) |
| 2. Event Driven Executive Utilities                    | (5719-UT5) |
| 3. Event Driven Executive Macro Library/Host           | (5740-LM4) |
| 4. Event Driven Executive Program Preparation Facility | (5719-XX4) |
| 5. Event Driven Executive Macro Library                | (5719-LM7) |

### **Basic Supervisor and Emulator (5719-XS3)**

Event Driven Executive application programs are made up of instructions coded in the Event Driven Language. At execution time, the assembled output of these instructions is passed to the emulator portion of the Supervisor/Emulator, and the Emulator links to the system routines required to perform the functions. The Supervisor portion of the Supervisor/Emulator manages system and I/O resources for application programs in execution.

### **Event Driven Executive Utilities (5719-UT5)**

The system utilities also operate under the control of the supervisor. They provide online, interactive support for a tailored supervisor generation, source module preparation, disk initialization, data set/volume maintenance, etc.

### **Event Driven Executive Macro Library/Host (5740-LM4)**

This is a set of libraries and procedures to be installed on a host System/370, so that Event Driven Executive or Series/1 assembler programs can be assembled on the host machine. The macros support all of the Event Driven Executive functions supported by the Event Driven Executive Program Preparation Facility (5719-XX4).

Prerequisites for host program preparation include:

- A binary synchronous communications line between the Series/1 and the host
- Use of either the S/370 Event Driven Executive Host Communications Facility IUP (5796-PGH) or the RJE utility supplied with Event Driven Executive Utilities (5719-UT5), for transfer of data sets between the two systems
- On the host, installation of the S/370 Program Preparation Facilities for Series/1 FDP (5798-NNQ)

### **Event Driven Executive Program Preparation Facility (5719-XX4)**

The Event Driven Executive Program Preparation Facility consists of programs which allow the user to assemble and link edit application programs concurrently with the execution of other programs (including other program preparation programs). The user can also reconfigure, assemble, and link edit custom supervisors online.

The Event Driven Executive assembler, \$EDXASM, (part of 5719-XX4) is used to assemble application programs written in the Event Driven language. As long as no Series/1 assembler language code is included in application source code (USER exit routines), this is the only assembler required for program preparation.

Licensed program 5719-ASA, (separately orderable program, not part of 5719-XX4) the Series/1 Macro Assembler, also runs under the Event Driven Executive system, and is used to assemble programs written in Series/1 assembler language. When installed under the Event Driven Executive, the Series/1 Macro Assembler is called \$\$S1ASM. If the Series/1 Macro Library (5719-LM7) is also installed, \$\$S1ASM may be used to assemble Event Driven Executive supervisors and programs written in the Event Driven Language, as well as programs written in Series/1 assembler language.

## Event Driven Executive Macro Library (5719-LM7)

This library contains the macro prototypes for the Event Driven instruction set, and all the macros necessary to build a supervisor tailored to a particular system configuration. This library is used when preparing programs using the Series/1 Macro Assembler \$\$1ASM (not required if system generation/program preparation is done with \$EDXASM).

### EVENT DRIVEN EXECUTIVE – AN OPERATIONAL OVERVIEW

The Event Driven Executive component that controls execution of user-written applications is the Supervisor/Emulator. It is a multi-programming supervisor, capable of controlling concurrent program execution.

The basic unit of work for the supervisor is an instruction. Instructions are combined to form tasks, each of which has an assigned priority, used by the supervisor to allocate system resources.

An application program may have more than one task (multitasking). Each task competes for system resources with every other task in the system, based on task priority. Each task runs independently of all other tasks.

Programs/tasks are made up of Event Driven Executive instructions that have been processed by an assembler and prepared for execution by the link/formatting system utilities. At execution time, the Supervisor/Emulator analyzes an instruction's assembled format, and links to the appropriate supervisor routine to perform the operation. Following the completion of each instruction, the supervisor processes the next sequential instruction in the highest priority task that is ready.

The Supervisor/Emulator occupies the lowest 10 to 40+ K bytes of Series/1 storage, depending on what support is included. The rest of storage is available for user application programs. Programs may be loaded by a terminal operator request, or by execution of a LOAD instruction in a currently executing program. Programs are loaded dynamically, using a relocating loader, into the smallest available area of storage of sufficient size to contain them.

Other functions/services performed by the supervisor include task dispatching (starting/ending tasks), I/O interrupt handling, program/task synchronization, and provision for inter-program communication via a global common area.

## Section 2. Instruction Format

### EVENT DRIVEN EXECUTIVE BASIC INSTRUCTION FORMAT

**OBJECTIVES:** After completing this topic, the student should be able to describe the basic format used in coding Event Driven Executive instructions.

### LANGUAGE SYNTAX/CODING CONVENTIONS

The Event Driven Executive instruction set was originally implemented as a macro library, using a macro assembler on the native or a host machine to process application source modules. \$EDXASM is an online Event Driven Executive language assembler, not a macro assembler, and does not utilize a macro library to process application source modules. Although macros are not used, macro assembler language syntax and coding conventions are still followed, thereby retaining compatibility with previous releases.

If required, Series/1 macro assembler language syntax/coding conventions may be reviewed in the Series/1 Macro Assembler Language Reference (SC34-0317).

## INSTRUCTION FORMAT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "General Instruction Format."

The basic Event Driven Executive instruction format is:

Label op opnd1,opnd2,...opndn,KEYWORD=,P1=,P2=,...Pn=

where

Label identifies the location of a particular instruction and can be referenced by other instructions.

op is the operation to be performed by the Series/1 (MOVE, ADD, etc.)

opnd1,opnd2,...opndn are positional operands. The meaning of each parameter or operand is defined by its position in the operand field of the instruction. The number of positional operands varies with each instruction type.

opnd1 is normally the "to" or target location.

opnd2 is normally the "from" or source location.

KEYWORD= are keyword operands. The keyword (PREC, RESULT, EVENT, etc.) specifies a particular parameter to be used in that instruction's execution.

P1=,etc are keyword operands that allow positional operand modification at execution time.

Figure 2-1 shows the relationship of the various parts of a source statement to the general instruction format. (The ADD instruction is discussed in detail in "Section 5. Data Manipulation", and is used here only to illustrate the basic instruction format.) In this example, three positional operands are used. FIELD is the name of the "to" or "target" location, DATA is the "from" or "source" location, and the third positional operand is the integer value "1", the "count" operand. A keyword operand, PREC= is also coded; in this case, the "S" indicates "single precision."

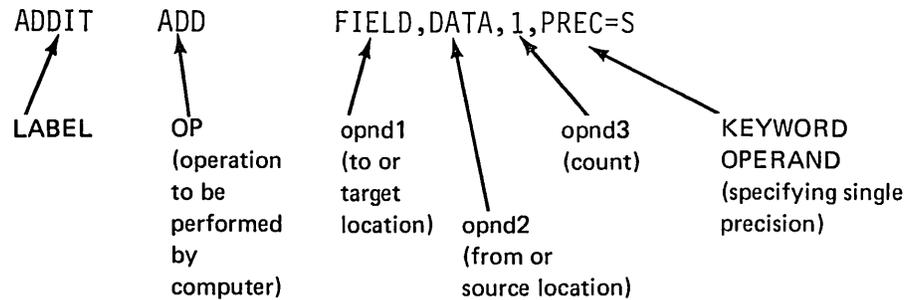


Figure 2-1. Source statement/general instruction format relationship

For the ADD instruction, the count and PREC = operands are not required; they have values to which they will default if not coded (the values coded in the illustration are, in fact, the default values for these operands). In the ADD, the "count" operand applies to the first positional operand only (the number of consecutive values, beginning at location FIELD, to which the value in DATA is to be added), and the "PREC =" operand, as coded, applies only to the first positional operand and the result (which is also the first operand, in this example).

Other instructions may not have a count or PREC= operand or, if they do, they may apply to other than the first positional operand. The general *syntax* of an Event Driven Executive instruction adheres to the basic format just discussed; the *meaning* of the operands, and the number of operands allowed differs depending on the instruction type.

This page intentionally left blank.

## INSTRUCTION FORMAT REVIEW EXERCISE – QUESTIONS

1. In the study guide, and in the reading assignment, the terms “operand” and “parameter” are both used. These terms are interchangeable, and both refer to labels/names/values in the operand field of an instruction.

True \_\_\_\_\_

False \_\_\_\_\_

2. In the operand field of an instruction, all positional operands used must precede (from left to right) any keyword operands used.

True \_\_\_\_\_

False \_\_\_\_\_

3. All instructions have the same number of positional operands, but the number of keyword operands varies from instruction to instruction.

True \_\_\_\_\_

False \_\_\_\_\_

4. In the operand field of an instruction, positional operands are separated by commas, but keyword operands may be separated by blanks or by commas.

True \_\_\_\_\_

False \_\_\_\_\_

5. The meaning of a positional operand, in a given instruction, is determined by its position (first, second, etc.), while the meaning of a keyword operand is determined by the keyword used.

True \_\_\_\_\_

False \_\_\_\_\_

6. Labels beginning with “\$” have a special meaning to the system, and are reserved for system use.

True \_\_\_\_\_

False \_\_\_\_\_

## INSTRUCTION FORMAT REVIEW EXERCISE – ANSWERS

1. True. Both terms are used interchangeably, throughout the study guide and the manuals. For example,

parameter one  
parameter 1  
first parameter  
parm1  
operand one  
operand 1  
first operand  
opnd1

are all used at one time or other to refer to the first positional operand in an operand field being discussed.

A possible area of confusion might be an instance when “parameter” is used to describe information passed to another program or a subroutine, rather than to reference an element of an operand field. Normal attention to the context in which the term is used will usually prevent any misunderstanding.

2. True. All positional operands must be coded before (to the left of) the first keyword operand. After all positional operands have been coded, multiple keyword operands may be coded in any sequence desired; all keywords are analyzed in light of the meaning of the keyword itself, rather than its position within the operand field.
3. False. Different instructions vary in the number of required positional operands (must be coded, no default), optional positional operands (will default to predetermined value if not coded), and required/optional keyword operands.
4. False. *All* operands, keyword or positional, are separated by commas, with no imbedded blanks allowed. When the first blank is detected, all further information is considered a comment.

In the situation where two or more optional positional operands are allowed, and you skip one and code the other, the skipped (defaulted) operand must be indicated by a comma if the coded operand follows it in position.

**Example:**

label op opnd1,opnd2,opnd3,opnd4  
REQUIRED OPTIONAL

***VALID OPERAND STRUCTURES***

opnd1,opnd2

REQUIRED OPERANDS ONLY – OPTIONAL OPERANDS (opnd3, opnd4) TAKE DEFAULT

opnd1,opnd2,opnd3

REQUIRED OPERANDS PLUS FIRST OPTIONAL OPERAND (opnd3) CODED – opnd4 TAKES DEFAULT VALUE

opnd1,opnd2,opnd3,opnd4

REQUIRED AND OPTIONAL OPERANDS CODED

opnd1,opnd2,,opnd4

REQUIRED AND LAST OPTIONAL OPERAND (opnd4) CODED, SKIPPED OPERAND (opnd3) INDICATED BY A COMMA

***INVALID OPERAND STRUCTURES***

opnd1,opnd2,opnd4

THE VALUE YOU THOUGHT YOU CODED FOR opnd4 WILL BE ASSIGNED TO opnd3, AND opnd4 WILL TAKE THE DEFAULT

5. True. Self explanatory.
6. True. There is no system enforced discipline preventing a user from defining storage locations with labels beginning with the "\$" character. However, because system defined functions/locations/resources have labels beginning with this character that may be referenced by operands in user-written instructions, confusion can be avoided if users restrict their own definitions to labels not beginning with "\$".

This page intentionally left blank.

## Section 3. Program/Tasks

**OBJECTIVES:** Upon successful completion of this topic, the student should be able to:

1. Describe programs and tasks as used in an Event Driven Executive System
2. Define an application program structure that fits system and application requirements
3. Use the Event Driven Executive program and task definition statements
4. Understand and use the task synchronization statements
5. Include operator attention routines in a program

### PROGRAM/TASK CONCEPTS AND STRUCTURE

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive System Guide (SC34-1702) "Program/Task Concepts and Structure" through "Multiple Program Structure." IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Task Control."

System resources in an Event Driven Executive system are allocated to tasks according to the priorities of the tasks. A task is a unit of work, defined by the application programmer. 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, the basic executable unit for the supervisor is the task.

Task priority is assigned by the application programmer when the task is coded. Valid priorities range between 0 and 511, with 0 being the highest possible priority, and 511 the lowest. Tasks with priorities between 0 and 255 execute on hardware level 2, and those between 256 and 511 on level 3.

## Single Task Program

For most applications, an elaborate program structure is not required, and programs will consist of a single task, as shown in Figure 3-1.

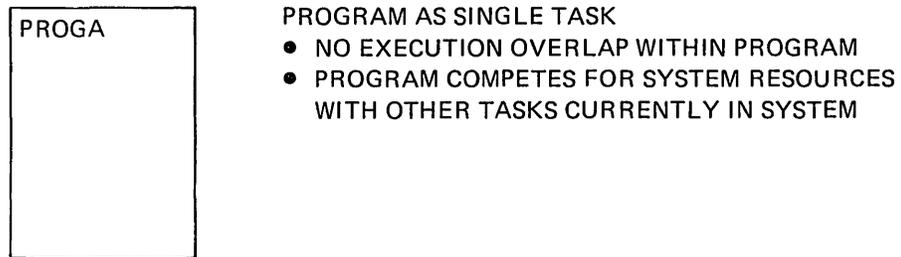


Figure 3-1. Single task program structure

Figure 3-2 is an example of the type of application that lends itself to the single task program structure. The job is sequential in nature, and will be waiting for operator input most of the time. There is no requirement for asynchronous execution of multiple functions or I/O overlap with processing, and nothing to be gained by a more complex structure.

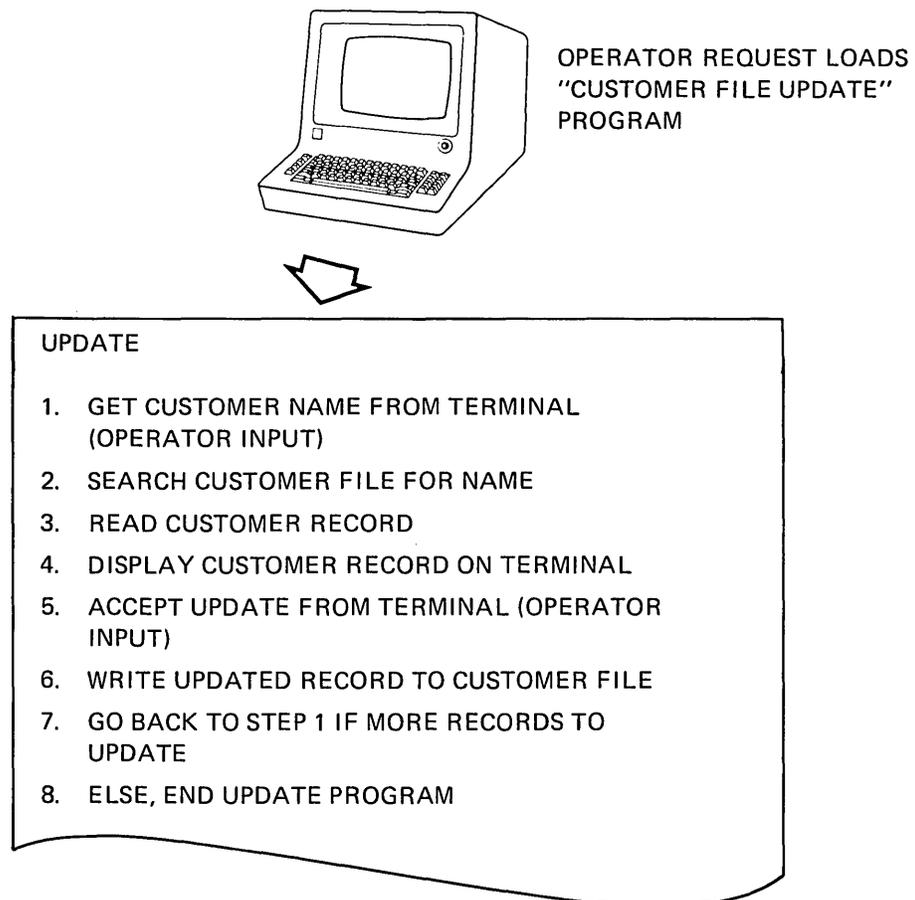
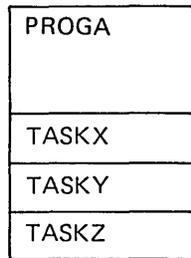


Figure 3-2. Single task application example

## Multiple Task Programs

Figure 3-3 illustrates a multitasking program structure. PROGA is started up by the system when the program is loaded, and is called the *PRIMARY TASK*. The other tasks shown will not start up until a user-coded command is executed that tells them to begin. *PRIMARY TASKS* go into execution as a result of the program's being loaded into storage, while initiation of *SECONDARY TASKS* is a user responsibility. Once in execution, all tasks within a program compete for system resources 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 resources based on task priority, regardless of which tasks are *PRIMARY* or *SECONDARY*.

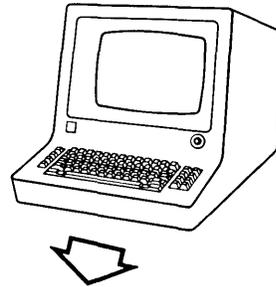


PROGRAM MADE UP OF MULTIPLE TASKS

- CONCURRENT (ASYNCHRONOUS) EXECUTION OF TASKS WITHIN PROGRAM
- TASKS COMPETE FOR SYSTEM RESOURCES WITH ALL OTHER TASKS CURRENTLY IN SYSTEM

Figure 3-3. Multitasking program structure

Figure 3-4 is an example of an application that makes use of multitasking. The program repetitively reads a group of Analog Input points, performs calculations on the data and stores the results in an output area on disk.



OPERATOR REQUEST  
LOADS "A/I DATA  
REDUCTION" PROGRAM

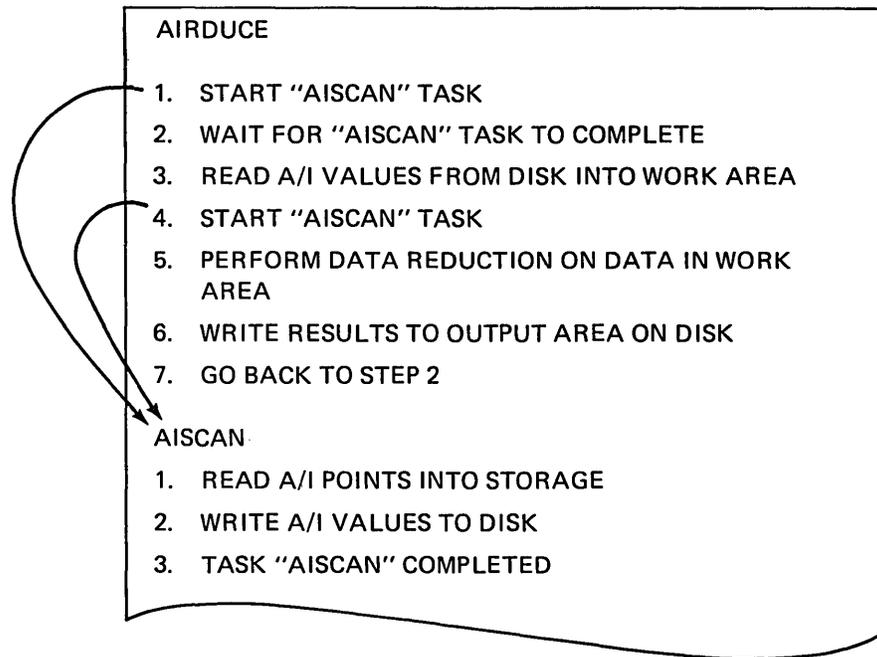


Figure 3-4. Multitasking application example

To take advantage of multitasking, the reading of the Analog Input points has been defined as a separate task, which also buffers the collected data to disk. When the program is loaded into storage, the supervisor starts up the primary task, AIRDUCE. The first step in AIRDUCE is to start up the secondary task AISCAN. AIRDUCE then waits for completion of the reading and buffering of the first set of Analog Input values.

When AISCAN completes, AIRDUCE starts up again, and retrieves the buffered data from disk. AISCAN is restarted and, while the first set of values is being processed, the second set is being collected; the two functions are overlapping.

## Multiple Program Structure

As already mentioned, an application program consists of a user-written collection of one or more tasks that has been prepared for execution and stored under a unique name on disk/diskette. A terminal operator can request that a program be loaded into storage and placed in execution by entering a request for the supervisor load utility \$L and supplying the program name.

Programs may also be loaded by executing a LOAD instruction in another program that is already in execution (use of the LOAD statement is discussed later in this section). When the supervisor receives a request to load a program, either from a terminal or a task already in execution, it finds the program on disk/diskette, finds a section of unused storage large enough to accommodate the program, loads the program from disk/diskette, relocates it into the storage area, and starts up the program's initial task. When a program completes execution, the supervisor releases the storage it occupied so that the area can be used to load other programs.

Because programs are dynamically relocated into storage as load requests are received, the size and structure of the programs can have an effect on system throughput. To illustrate this, assume there is a payroll application consisting of the following functions:

Function	Description
<b>SORT</b>	Separate part-time hourly, full-time hourly, and salaried employee data into three separate files.
<b>PART-TIME WAGES</b>	Process all records in part-time employee file
<b>FULL-TIME WAGES</b>	Process all records in full-time employee file
<b>SALARIED WAGES</b>	Process all records in salaried employee file
<b>WRITE CHECKS</b>	Print checks for all employees

Although the payroll job just described is a fairly straightforward application, which could be coded as a single program, there may be valid reasons for breaking it up into multiple programs. One consideration is the size of a program, in relation to the storage available on the system and the number and size of other programs that may need to run concurrently. If the size of PAYROLL in relation to the total storage available for user programs is as depicted in Figure 3-5, you can see that, once PAYROLL is loaded, little storage will be left for loading other programs.

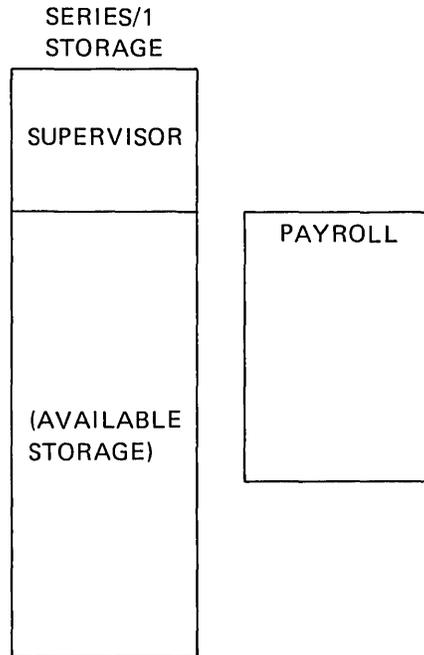


Figure 3-5. Program structure

Conversely, if other programs are already in execution when the load of PAYROLL is requested, there may be some delay before enough contiguous storage to accommodate so large a program becomes available and the load can again be attempted.

Below is a redefinition of the payroll application with each function coded as a separate program.

Program Name	Description
SORTIME	Separate part-time hourly, full-time hourly, and salaried employee data into three separate files
PARTIME	Process all records in part-time employee file
FULLTIME	Process all records in full-time employee file
SALTIME	Process all records in salaried employee file
CHECKS	Print checks for all employees

As can be seen in Figure 3-6, each of the programs is now much smaller than the entire PAYROLL program. As each program completes execution, it would request the load of the succeeding program. The probability of there being enough storage to load other applications is greatly increased, and the chance of having to wait for storage to become available so that you can again attempt to load a program there was previously no room for, is reduced.

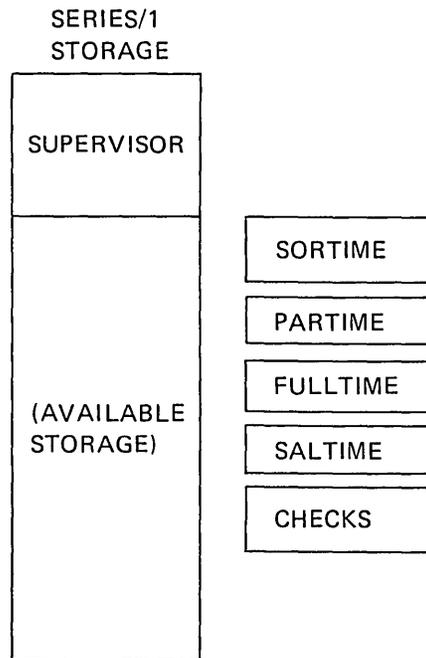


Figure 3-6. Program structure

If system activity were very high (several other applications in concurrent execution), a lack of contiguous storage availability could still cause some difficulty in the loading of the next sequential program. In a payroll application, this is acceptable, because it is not "time-critical"; a delay in execution of a succeeding step will not invalidate the final result.

### Overlay Program Structure

Some applications *are* time constrained; for example, those involving the processing of data acquired in realtime, where a delay in execution might result in data being lost or overwritten. This type of application must have a reasonable expectation of being loaded quickly when requested and, once loaded, of running to completion with minimal delay.

Coding a time-critical application as a single program ensures rapid execution, once it is loaded, but, if the program is large, the same problems exist as in the single-program payroll application (possible delay in load due to large amount of storage required; tying up system once loaded). Breaking up the program into separate programs takes care of the problem of size, but the requirement for nearly continuous execution once in operation, is still not met. Again, the level of activity within the system could result in a delay in loading the next in a sequence of programs, a condition that cannot be tolerated in this type of application.

Using the OVERLAY PROGRAM technique, both the requirement for a reasonable sized program and minimum execution delay can be met. In Figure 3-7, the application is split into separate programs.

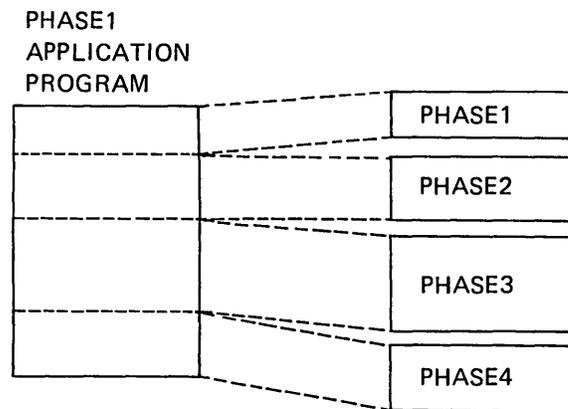
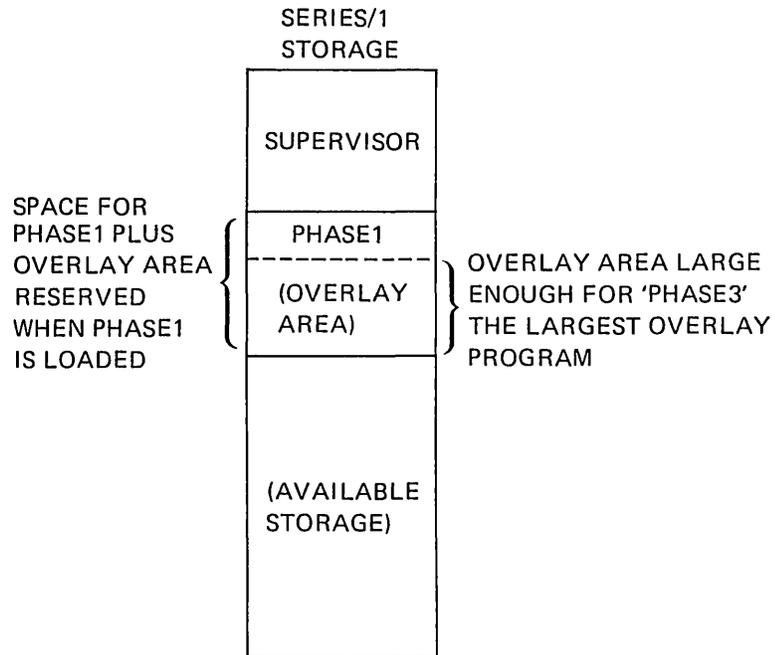


Figure 3-7. Program overlays

PHASE1 is the initial program, and will load PHASE2, PHASE3, and PHASE4, as required. PHASE2, PHASE3, and PHASE4 are defined as OVERLAY PROGRAMS. When PHASE1 is loaded, the loader recognizes that overlay programs are referenced. The loader looks at each program that is designated as an overlay, and then reserves enough storage to hold PHASE1 plus the largest overlay program.



**Figure 3-8. Program overlays**

When PHASE1 is loaded and in execution, and requests that PHASE2 be loaded, the system immediately loads PHASE2 into the overlay area already reserved and starts it into execution. There is no contention for the storage in the overlay area with other applications waiting to be loaded, because the overlay area is reserved for the exclusive use of PHASE1 overlay programs.

As each overlay program completes, PHASE1 loads the next, until all required programs have run. When PHASE1 terminates execution, the storage reserved for both PHASE1 and the overlay area is released.

To summarize, application program structure (single program/multiple programs/overlays) and task structure within programs (single task/multitasking) is determined by

1. type of application (time/non-time critical)
2. size of application
3. system storage size
4. operating environment (system activity/loading)

In general, a user should choose the simplest structure that will support the application's requirements.



All Event Driven Executive programs must end with an ENDPROG statement, followed by an END. These two statements must be the last two statements in the program.

Tasks within programs (other than the primary task) are defined by the TASK statement, and must end with the ENDTASK statement. The TASK statement performs the same functions for a task that the PROGRAM statement did for a program except that the data files and overlay programs defined in the PROGRAM statement apply for all tasks defined in that program, and are not specified in the TASK statement.

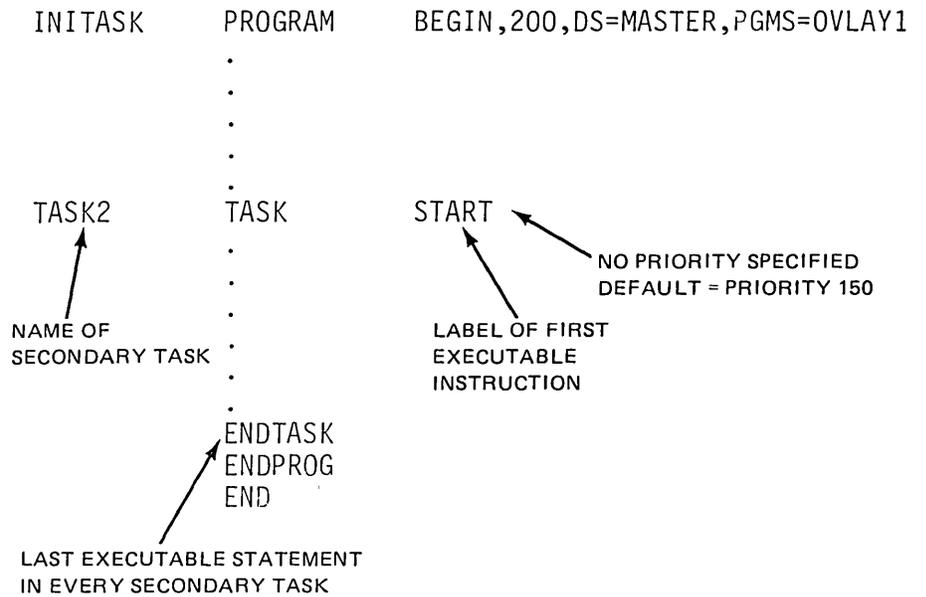


Figure 3-10. Task definition

## PROGRAM/TASK EXECUTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706) "PROGSTOP", "LOAD", "WAIT", "ECB", "ATTACH."

### Program Loading

Event Driven Executive programs are readied for execution at the time they are loaded into storage from disk or diskette (a given program will not immediately go into execution unless its primary task has a higher priority than other currently executing tasks). Programs are loaded by a terminal operator, using the \$L operator command, or by execution of a LOAD statement in a task already in execution. In both cases, the program to be loaded is referenced by the name under which it is stored on disk/diskette, and is either entered by a terminal operator, or specified as a LOAD statement operand.

*Note:* The name of a program on disk has no relationship to the name of that program's primary task. Illustrations in this study guide frequently show both names the same, but this is not a requirement of the system.

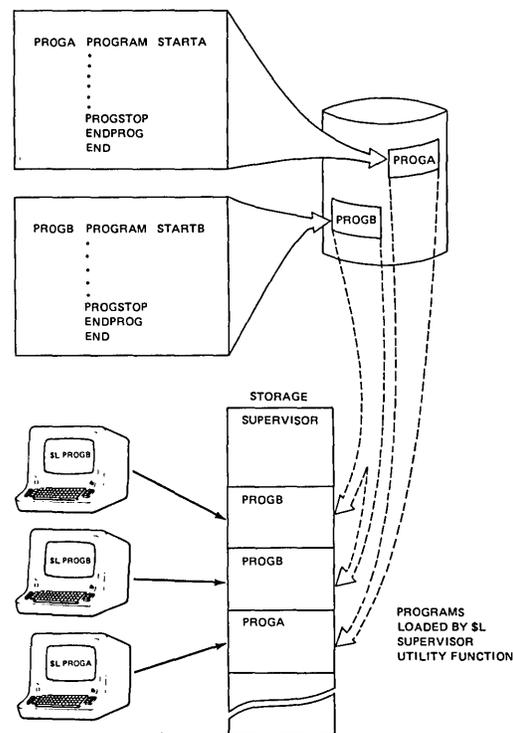


Figure 3-11. Program loading from terminal

As shown in Figure 3-11, copies of the same program may be in storage and active at the same time. The single copy of a program on disk/diskette may be loaded as a separate program from one or more terminals (as shown) as a separate program from one or more programs already executing, or as an overlay by a currently executing program or programs.

Figure 3-12 is a simple example of one program loading another. The program consists of the single task INITASK, which will start execution at location BEGIN. No priority is coded on the PROGRAM statement, so this program will run at the default priority of 150.

```
INITASK      PROGRAM      BEGIN
              .
              .
              .
              .
              .
BEGIN        LOAD          PTHREE
              .
              .
END          PROGSTOP
              ENDPROG
              END
```

Figure 3-12. LOAD statement

User disk/diskette I/O will not be performed in this program (DS= not coded in PROGRAM statement), and no overlay programs will be loaded by this program (PGMS= not coded).

Execution of the LOAD statement at location BEGIN requests that a program named PTHREE be loaded into storage and readied for execution. The loading program will wait for the completion of the attempt to load PTHREE before continuing execution.

The last statement to be executed in the loading program is the PROGSTOP at location END. The PROGSTOP statement must be the last executable statement in all programs. When PROGSTOP is executed, the supervisor is notified that this program's primary task is to be detached (made not active), various system resources that were assigned to this program can now be made available to other tasks, and the storage occupied by this program can be released for the loading of other programs.

In the oversimplified example shown in Figure 3-12, the loading task does not check to make sure the load operation was successful. In actual practice, the user would want to know if the operation failed, and if it did, the reason for the failure.

In Figure 3-13, the program location ABORT is specified in the ERROR= keyword operand. If the load is successful, execution continues with the statement following the LOAD. If the load operation fails, control is transferred to the location specified by the ERROR= keyword operand. In this example, ABORT is the label on a PROGSTOP statement and failure of the load operation would result in termination of the loading task. (In actual application programs, error routines are likely to be much more complex.)

```

INITASK      PROGRAM      BEGIN
              .
              .
              .
              .
              .
BEGIN        LOAD          PTHREE ,ERROR=ABORT
              .
              .
              .
              .
              .
ABORT       PROGSTOP
              ENDPROG
              END

```

Figure 3-13. LOAD statement

Every task has a Task Control Block (TCB) associated with it. A task's TCB is automatically generated during the program preparation process when a task definition statement is encountered. A TCB consists of those pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of the task in storage.

The first word of a task's TCB is used by the supervisor to pass information from the system to the task, regarding the outcome of various operations the task has initiated. Depending on what operation was attempted, the value set in the first word of the TCB by the supervisor could indicate an arithmetic exception condition, the result of an attempted I/O operation, or, as in Figure 3-13, a load operation completion code.

When a TCB is generated, the location of the first word is assigned the label on the task definition statement: the "name" of the task. In this study guide, and in Event Driven Executive reference documentation, this label is referred to as the "taskname," and the first word of the TCB is called the "task code word." In Figure 3-13, the task code word would be referenced by the taskname INITASK. If ABORT (specified in ERROR= keyword operand of LOAD statement) were the label of a user-written error routine, instructions in that routine could get the load operation completion code by using INITASK to locate the task code word. Appropriate operator messages could then be printed out or alternative actions taken, based on the precise meaning of the completion code.

At this point, the instructions required to examine the task code word have not been discussed; however there will be examples illustrating this technique in later sections of this course.

## Program Synchronization

Assuming the LOAD operation was successful, and PTHREE does go into execution, the loading program illustrated in Figure 3-13 has no way of telling when PTHREE finishes execution. For some applications, there is no need for a loading program to be notified of a loaded program's completion, but there are cases where synchronizing the execution of programs or tasks is required. This can be accomplished by defining an event, and waiting for that event to happen.

The "wait on event" facility is a signalling mechanism whereby a task or program can be notified when a certain event has occurred, and can wait or suspend execution until it does occur. Events include such things as the expiration of a time delay, completion of an I/O operation, or termination of a task or program. Events may be user defined or, for some frequently required functions, may be predefined by the system.

Completion of program execution is a predefined event, invoked by coding the EVENT= keyword operand in the LOAD statement. In Figure 3-14, the event has been named DONE3, which is also the label of an Event Control Block (ECB) that is used by the supervisor to keep track of whether the event has or has not occurred.

```

INITASK      PROGRAM      BEGIN
.
.
.
.
.
BEGIN        LOAD          PTHREE ,EVENT=DONE3 ,ERROR=ABORT
.
.
.
.
.
                WAIT          DONE3
.
.
.
.
.
ABORT        PROGSTOP
DONE3        ECB
                ENDPROG
                END

```

Figure 3-14. LOAD statement

*Note:* If preparing programs using \$S1ASM, the Series/1 Macro Assembler, coding the EVENT= keyword operand in a LOAD statement causes an ECB with the proper label to be automatically generated. When preparing programs using the Event Driven language assembler \$EDXASM, the ECB must be coded, as shown in Figure 3-14.

When the LOAD statement is executed, the supervisor recognizes that an event has been defined in the EVENT= keyword operand. The supervisor finds the ECB named DONE3, and sets it to indicate that the event has not occurred.

After PTHREE has been loaded, both PTHREE and the loading program are in execution concurrently. Eventually PTHREE will complete execution (execute a PROGSTOP) and, at that time, the supervisor will set the ECB at location DONE3 to indicate that the event has occurred.

When the WAIT statement in the loading program is executed, the supervisor will see that the waited-on event is DONE3. The supervisor checks the ECB at location DONE3 to see if the event has occurred. If it has, execution continues with the next statement following the WAIT. If it has not, the loading program is placed in a wait state, and execution will not resume until PTHREE completes. When an event occurs, and the associated ECB is set to indicate that it has occurred, the supervisor also checks to see if there are any tasks in wait state, waiting on that event. If there are, the supervisor changes them to the ready state, and they resume normal execution, based on priority.

For examples of how user-written events are defined and used, see the discussion titled "WAIT/POST" later in this section.

One instance where waiting on a "completion of execution" event such as was just described *must* be done is when a program loads an overlay. It is a user responsibility to ensure that a program that loads an overlay program does *not* execute a PROGSTOP until the overlay program has completed execution.

If a program has loaded an overlay program that is now executing, and the loading program issues a PROGSTOP, the storage occupied by the loading program and the overlay area is released to the system, and made available for loading other programs. Although the overlay area contains a program still in execution, the loader believes the storage is available, and may, in response to a load request, load another program into the same area, with completely unpredictable results.

In Figure 3-15, PTHREE is defined as an overlay program in the PGMS= operand of the PROGRAM statement. Up to nine overlay programs may be defined in a PGMS= list.

```

INITASK      PROGRAM      BEGIN,PGMS=PTHREE
              .
              .
              .
              .
              .
BEGIN        LOAD          PGM1,EVENT=DONE3,ERROR=ABORT
              .
              .
              .
              .
              .
ABORT        WAIT          DONE3
DONE3        PROGSTOP
              ECB
              ENDPROG
              END

```

Figure 3-15. LOAD statement

The LOAD statement requests the load of PGM1. This is a positional keyword reference to the PGMS= list in the PROGRAM statement. If multiple overlay programs were defined in the PGMS= operand, and you wished to load the second program in the list, the LOAD statement would be coded to load PGM2; for the third program, PGM3, and so on up to the maximum of PGM9.

Note that the EVENT= keyword operand in the load statement is coded, and that the loading program waits for completion of the overlay program before issuing a PROGSTOP.

A program's primary task is started into execution (placed in a ready state) by the system at the time the program is loaded. Secondary tasks within a program are readied for execution by an ATTACH instruction, issued from the primary task or another secondary task previously attached and running.

In Figure 3-16, a secondary task called TASK1 is defined. TASK1 will be started up by the ATTACH in the primary task, at location BEGIN. Once TASK1 has been attached, TASK1 and INITASK, the primary task, execute concurrently and independently.



Now assume that the program has been loaded, and the system has attached the primary task, INITASK. Execution starts at location BEGIN. This statement requests the load of overlay program PFIVE, because PFIVE is the second program in the PGMS= list of the PROGRAM statement, and the LOAD statement specifies PGM2. If the load of this first overlay fails, the ERROR= operand of the LOAD statement will cause a transfer of control to location OUT5BAD, the label of the PROGSTOP, and execution will terminate.

```

INITASK      PROGRAM      BEGIN,100,PGMS=(PTHREE,PFIVE)
.
.
BEGIN        LOAD          PGM2,EVENT=DONE5,ERROR=OUT5BAD
L4           LOAD          PFOUR
.
.
.
A1           ATTACH       TASK1
.
.
.
W5           WAIT         DONE5
L3           LOAD          PGM1,EVENT=DONE3,ERROR=OUT3BAD
.
.
.
W3           WAIT         DONE3
OUT3BAD     WAIT         TASKDONE
OUT5BAD     PROGSTOP
.
DONE5       ECB
DONE3       ECB
.
.
TASK1       TASK          START,EVENT=TASKDONE
.
.
.
.
.
ENDTASK
ENDPROG
END

```

Figure 3-17. Task/program synchronization

If PFIVE loads properly, the next statement executed would be the LOAD instruction at location L4. This statement requests that program PFOUR be loaded into whatever storage is available (not in overlay area). As it is coded here, any errors encountered in attempting to load PFOUR will be ignored, and execution will continue with the statement following the LOAD.

At location A1, the primary task attaches the task defined at location TASK1, at a priority of 150 (default taken, and no override coded in the ATTACH). At this point, the primary task INITASK is executing, the secondary task TASK1 is executing, the primary task of PFIVE, and any secondary tasks it attached are running in the overlay area, and if PFOUR loaded successfully, it is also in execution.

Before attempting to load overlay program PTHREE (LOAD statement at location L3), a WAIT at location W5 is executed, waiting on the completion of execution event defined in the LOAD statement which previously loaded PFIVE (EVENT=DONE5). If PFIVE has not finished, the execution of INITASK is suspended at this point. When PFIVE completes, or if PFIVE were already through when the WAIT at W5 was issued, the LOAD at location L3 is attempted.

This is a load of PTHREE, the first (PGM1) overlay program defined in the PGMS= list of the PROGRAM statement. Notice that if the load operation fails, the ERROR= operand of the LOAD statement would cause a transfer of control to location OUT3BAD, which is a WAIT for the completion of TASK1, rather than to OUT5BAD, the PROGSTOP. If the load of PTHREE were unsuccessful, the primary task is assured that no program is executing in the overlay area, but the secondary task TASK1 could still be in operation. Any overlay program in execution, and all attached tasks, must run to completion before PROGSTOP is executed by the primary task.

*Note:* In the figures in the study guide, no user-coded ECBs are shown for event control blocks named in the EVENT= operands of TASK statements. When programs are prepared using the Event Driven language assembler \$EDXASM, the system will automatically generate the required ECB with the TCB created by the TASK statement, and a user-coded ECB is not allowed (will cause assembly errors). Users preparing programs under the Series/1 macro assembler may also allow the system to assign the ECB, or may code an ECB of that name, and the system will use the explicitly coded ECB instead of assigning one.

If disk or diskette I/O is used in a program, the data sets to be accessed must be defined in the PROGRAM statement's DS= operand, in much the same manner as overlay programs are specified using PGMS=. This topic will be discussed in the DISK I/O section of this study guide.

## QUEUABLE RESOURCES

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "ENQ", "DEQ."

A resource is a physical or logical entity within the system. Examples of resources include a subroutine or data area existing within a particular program, or perhaps a data set or I/O device known broadly across the system.

A shared resource is one that may be required by multiple tasks at the same time. For instance, a table of constants might be referenced from two or more asynchronously executing tasks within a program. Since, by definition, the values in the table are "constant" (not being altered by the tasks using them), access to the table (resource) is unrestricted.

Unrestricted access to some shared resources may have undesirable results. As an example, if a program were printing a report on a printer, and other programs had free access to the printer resource, the report could end up with printed output from the other programs interspersed with report lines. In this case, the printer is a shared resource, but is also what is called a serially reusable resource; one that should be used by only one task at a time.

The ENQ/DEQ instructions provide a mechanism by which user tasks may gain exclusive use of a serially reusable shared resource, and retain control over that resource until explicitly releasing it for use by other tasks.

Figure 3-18 is an example of how queueable resources are defined and used. The program consists of the primary task INITASK, and two secondary tasks, TASKA and TASKB. Assume that both TASKA and TASKB have a requirement for a 500-word work area.

Instead of putting a 500-word work area in both TASKA and TASKB, the programmer has chosen to save some storage, and define only one work area. This single work area is designated as a queueable resource, and will be shared by TASKA and TASKB, using the ENQ and DEQ instructions.

The 500-word work area is defined in the DATA statement at location CALCTABL (DATA statements are discussed fully in a later section). The Queue Control Block for this resource is defined in the QCB statement at location CALCQ.

*Note:* If preparing programs using the Series/1 macro assembler, coding an ENQ statement causes the automatic generation of a QCB with the same label as specified in the operand of the ENQ. When preparing programs using the online assembler (\$EDXASM), users must code the QCB; it is not automatically generated.

INITASK	PROGRAM	STARTUP
	.	
STARTUP	ATTACH	TASKA
	ATTACH	TASKB
	.	
	.	
W1	WAIT	AFINISH
W2	WAIT	BFINISH
	PROGSTOP	
	.	
	.	
CALCTABL	DATA	500F'0'
	.	
	.	
CALCQ	QCB	
	.	
TASKA	TASK	ASTART,EVENT=AFINISH
ASTART	ENQ	CALCQ
	.	
	.	
	.	
	DEQ	CALCQ
	ENDTASK	
	.	
	.	
TASKB	TASK	BSTART,EVENT=BFINISH
BSTART	ENQ	CALCQ
	.	
	.	
	.	
	DEQ	CALCQ
	ENDTASK	
	ENDPROG	
	END	

Figure 3-18. ENQ/DEQ/QCB

When the program begins execution, the primary task attaches both TASKA and TASKB. TASKA and TASKB have agreed to the convention that any time either of them needs to use the work area CALCTABL, they will enqueue that resource by issuing an ENQ instruction referencing the QCB called CALCQ. Assuming that TASKA issues the ENQ first, the supervisor checks the QCB at CALCQ, finds that no other task is currently enqueued, and gives exclusive control of the work area to TASKA. TASKA can now use CALCTABL without fear of TASKB altering its contents in mid-execution.

While TASKA has the work area enqueued, TASKB, which is also in execution, attempts to gain control of the work area by issuing its own ENQ of CALCQ. The supervisor checks the QCB, finds that TASKA is already using the resource represented by CALCQ, and therefore places TASKB in the wait state, waiting upon availability of the requested resource.

When TASKA is finished with the work area, it issues a DEQ of CALCQ. The supervisor checks the QCB, and finds that TASKB is waiting on that resource. TASKB is placed back in the ready state, and the QCB is changed to indicate TASKB's "ownership" of the resource represented by CALCQ.

An additional operand, not shown in the example, may be coded on the ENQ statement. This is the keyword operand BUSY=. It would be coded if, when attempting to ENQ a resource and the resource was busy (enqueued by another task), you did not want to suspend, waiting for the resource to be dequeued. You may code the label of an instruction in the BUSY= operand (BUSY=label), and control will be transferred to that location if the resource is already enqueued when your task tries to ENQ it.

Note that ENQ/DEQ provides protection from simultaneous access of a serially reusable resource only if all users requiring the resource agree to employ it. In the example in Figure 3-18, if one of the two tasks were to use the CALCTABL work area without first enqueueing for it, neither the supervisor nor the other task has any way of detecting or preventing it.

## WAIT/POST OPERATION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706) "POST", "RESET", "WAIT."

Figures 3-14 through 3-17 illustrated how a program or task can synchronize execution with a loaded program or attached task by using a WAIT on the ECB named in the associated LOAD or TASK statement's EVENT= operand. The EVENT= operand is a convenient means of synchronizing the execution termination sequence of loading and loaded programs or attaching and attached tasks, but programs and tasks often require synchronization at other points in their execution. This can be accomplished through user-defined events, and the WAIT/POST mechanism.

In the example in Figure 3-19, assume that the primary task, WAITPOST, at some point in its execution, requires a certain set of numeric values in order to continue. These values are the result of the execution of a calculation routine in XTASK, an attached secondary task, the primary task must therefore make sure that the calculation routine in XTASK has been executed, before proceeding with its own execution.

The primary task could wait on the EVENT= operand in the TASK statement XTASK (EVENT=TASKDONE), and be assured that the required values had been calculated. This method would work, but the entire secondary task would have to run to completion before WAITPOST could resume execution. Depending on what else XTASK has to do in addition to the calculation routine, there could be a considerable amount of time in which the required values were ready for use, but WAITPOST is still in a wait state.

Defining the completion of the calculation routine in XTASK as a user event allows XTASK to signal the primary task as soon as the required values have been generated. The event is called CALCDONE, and an ECB of that name is coded. ECBs for user-defined events are initially set up to indicate "event occurred." A WAIT issued against such an ECB will act as though the event has happened (fall through). Therefore, a RESET of the ECB must be executed before a WAIT is issued against it. The RESET instruction at location INITGO sets the ECB to indicate "event has not occurred."

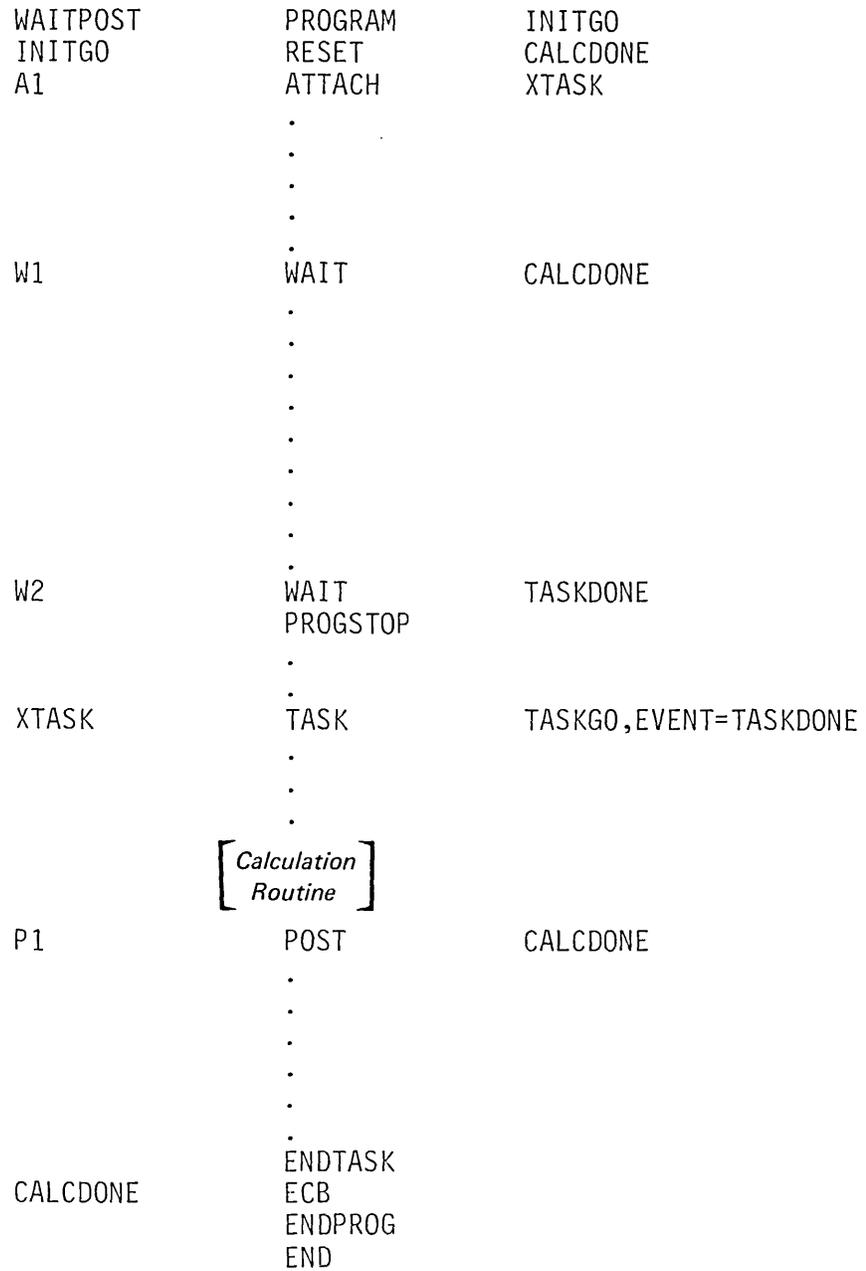


Figure 3-19. WAIT/POST

In the example, execution begins with the RESET command at location INITGO, which changes the ECB at CALCDONE from its initial indication of "event occurred" to "event has not occurred." At location A1, the secondary task XTASK is attached. WAITPOST and XTASK are now in concurrent but asynchronous execution. When XTASK finishes calculating the values required by the primary task, the POST instruction at location P1 is executed, and the ECB at location CALCDONE is set to indicate "event occurred."

At the time the POST is issued, the supervisor checks to see if there are any tasks waiting on this event. If the WAIT at W1 had already executed, the primary task would now be in a wait state, and the supervisor would place WAITPOST back in a ready state. If the WAIT had not yet occurred, WAITPOST would continue executing until it was encountered. When the WAIT was issued, the supervisor would check CALCDONE, and, finding the event already complete, would allow WAITPOST to continue execution.

The instructions following the WAIT at W1 in the primary task, and the instructions following the POST at P1 in the secondary task can now continue executing concurrently; the primary task did not have to wait until the secondary task terminated before using the required values. (Notice that the proper termination sequence for an attaching and an attached task is still necessary, and is provided for in the example by the WAIT on EVENT=TASKDONE at location W2.)

The RESET instruction is used with user-defined events. System-defined events, such as those declared in the EVENT= operand of LOAD or TASK statements, are automatically initialized by the system. The use of RESET with a system-defined event may result in improper or unpredictable operation.

*Note:* When preparing programs using the Series/1 macro assembler, declaring an event name in the operand of a POST statement results in the automatic generation of an ECB of the same name. Users of the Event Driven language assembler \$EDXASM must code an ECB with a label matching the name in the POST operand; ECB generation is not automatic.

## ATTENTION LISTS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "ATTNLIST", "ENDATTN."

The ATTNLIST capability provides a means for an operator to communicate with a program using a terminal. The ATTNLIST statement is used to specify operand pairs, each pair consisting of a 1- to 8-character operator command, and a label in the user program, which will receive control when that operator command is entered.

In the example in Figure 3-20, the ATTNLIST statement defines a single operand pair, STOP, XTHREE. (Note that ATTNLIST, like ECB and QCB, is not an executable statement, and must not be coded within an executable code sequence.) The first "name" in the operand pair defines an operator command to be entered from a terminal, and the second is the label of the instruction in the user program that will be executed when that command is entered.



Attention routines usually set a program indicator that can be checked by the user task; execution-time decisions (end execution, restart the program, load another program) can then be made, based upon the value in the indicator. The instructions necessary to set storage locations (program indicators) or check them for specific values have not yet been discussed, and are therefore not shown in Figure 3-20. For further discussion and complete examples, see the topic "Operator Control of Program Execution" in "Section 11. Terminal I/O."

**PROGRAMS/TASKS –REVIEW EXERCISE –QUESTIONS**

1. Most applications can be programmed as a single task. What type of application would justify the use of the more complex multitasking structure illustrated in Figure 3-3?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

2. What are the advantages of loading a program as an overlay, rather than just loading it into available storage?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

3. What disadvantages are there to the overlay program structure?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

4. How does a program's primary task get started up?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

5. What statement must be executed to release the storage occupied by a program?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

This page intentionally left blank.

6. Fill in the blanks in the following paragraph, using words or phrases from the list below. (Some items in the list may be used more than once, and some not at all.)

- |                    |                 |
|--------------------|-----------------|
| a. ENDTASK         | f. PROGRAM      |
| b. ATTACH          | g. ENDPROG      |
| c. entry point     | h. PROGSTOP     |
| d. TASK            | i. END          |
| e. shared resource | j. primary task |

"The first statement in all programs is the \_\_\_\_\_ statement. The label of this statement establishes the name of the program's \_\_\_\_\_. The last two statements in every program must be \_\_\_\_\_ and \_\_\_\_\_. The \_\_\_\_\_ statement must be the last statement in a primary task to be executed. The first statement in a secondary task is the \_\_\_\_\_ statement. The statement which defines the end of a secondary task, and which is also the last to execute, is \_\_\_\_\_."

7. What is the purpose of ENQ/DEQ and the QCB?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

8. The proper execution termination sequence of loading/loaded programs and attaching/attached tasks is an automatic function of the Event Driven Execution supervisor.

True \_\_\_\_\_

False \_\_\_\_\_

9. In Figure 3-20, assuming the program is in storage and executing, and the operator enters QUIT after pressing the Attention key on the terminal, which of the following would be true?

- a. The program would immediately execute the PROGSTOP instruction, terminating execution.
- b. The program would execute the attention routine at location XTHREE.
- c. The entry would not affect program execution.
- d. The program would be placed in a wait state, waiting for the operator to enter XTHREE.
- e. None of the above.

## PROGRAMS/TASKS REVIEW EXERCISES – ANSWERS

1. A user might consider multitasking where speed of execution is of primary importance, and the nature of the job is such that certain functions may be overlapped (i.e., I/O and processing).
2. When loading an overlay program, the loading program is assured that space is available, because it is reserved at the time the loading program itself is loaded. Also, the load of an overlay program is faster than the load of the same program into available storage would be. This is because information about the overlay program which the loader requires in order to load it is looked up at the time the loading program is loaded, and not at the time the LOAD command is executed, as is the case when loading a non-overlay program.
3. The storage occupied by a program that loads overlays is always equal to the size of the loading program plus the size of the largest overlay. If the loading program executes without requiring any overlays, the overlay area, although unused, is still unavailable to the rest of the system.
4. The primary task is "attached" (made ready for execution) by the system (actually the loader) at the time a program is loaded to storage. Activation of secondary tasks is a user responsibility, accomplished by execution of ATTACH instructions in already running primary or secondary tasks.
5. Execution of PROGSTOP makes the storage now occupied by a program available to the system, and terminates (detaches) the program's primary task.
6. The first statement in all programs is the f) PROGRAM statement. The label of this statement establishes the name of the program's j) primary task. The last two statements in every program must be g) ENDPROG and i) END. The h) PROGSTOP statement must be the last statement in a primary task to be executed. The first statement in a secondary task is the d) TASK statement. The statement which defines the end of a secondary task, and which is also the last to execute, is a) ENDTASK.
7. ENQ and DEQ are used to protect against the concurrent use of a serially reusable shared resource by asynchronously executing tasks.
8. FALSE. This is a user responsibility. The system provides the WAIT/EVENT=/ECB to accomplish it (and WAIT/POST for user events), but the user must code the required statements.
9. Choice c. is correct. The ATTNLIST in Figure 3-20 defines the character string STOP as the operator input required to execute the attention routine at location XTHREE. Any other entry is ignored.

## Section 4. Data Definition

**OBJECTIVES:** After completing this section, the student should be able to:

1. Define data constants for the following data types:
  - a. EBCDIC
  - b. Hexadecimal
  - c. Binary
  - d. Fixed Point
  - e. Floating Point
  - f. Address Constant
2. Define symbolic data areas using the TEXT and BUFFER statements
3. Define a text message using the TEXT statement

Data definition statements are used to define arithmetic values or character strings (constants and messages) and to reserve areas of storage for use during program execution (I/O buffers, work areas).

### DATA STATEMENT

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Language Reference (SC34-1706) "DATA."

The DATA statement is the Event Driven Executive equivalent of the Series/1 assembler language Define Constant (DC) statement. Although all of the examples in this study guide use DATA statements, DC statements could be coded in their place, with the same results.

*Note:* This is the only instance where a Series/1 assembler language statement may be coded in an Event Driven Executive program without employing the USER statement. See "Section 7. Program Control" of this study guide for discussion and examples of the USER instruction.

The format for the DATA statement is shown in Figure 4-1.

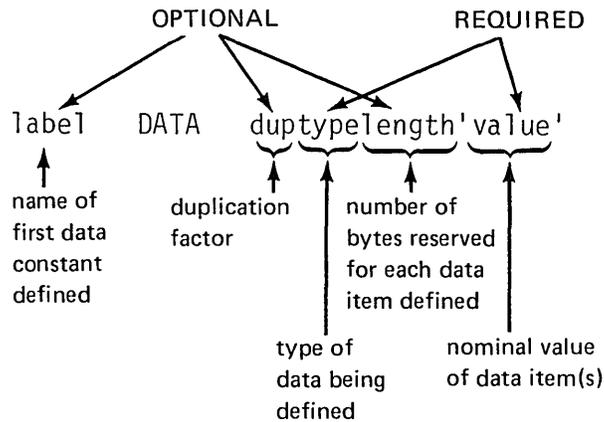


Figure 4-1. Data statement

The DATA statement is made up of at least two ("type" and "value") or as many as four parts. The first three parts ("dup," "type," and "length") are data descriptors or modifiers. The last part, "value," is coded with the actual data being defined. All parts of the DATA statement are coded contiguously; no separators, such as blanks or commas, are allowed.

- dup            duplication factor. This optional operand modifier is coded as an integer value, indicating how many repetitions of the data item defined by the rest of the operand should be generated. If not coded, dup defaults to 1 (one).
- type           data type. This defines the type of data being defined, and must be coded in every DATA statement. Nine data types are supported by the system, each one represented by a different alpha character. The type of data desired is indicated by coding the appropriate character in the type portion of the operand.
- length        number of bytes to be used for each data item. The length modifier is supported for only hexadecimal (data type X) and EBCDIC (data type C) data, and is optional for those. Every data type (including hexadecimal and EBCDIC) has an implicit length associated with it. This length is the number of bytes required to hold the assembled output of the data constant defined. For example, every EBCDIC character is represented by an 8-bit (one byte) binary code. Therefore, when EBCDIC character strings are defined in DATA statements, the assembled output requires one storage location (one byte) for each character in the string (upper example in Figure 4-2). The length modifier overrides this implicit length of one byte per character. The assembled output of the character string is placed in the number of bytes specified in the length modifier, with truncation or padding of the character string if required.

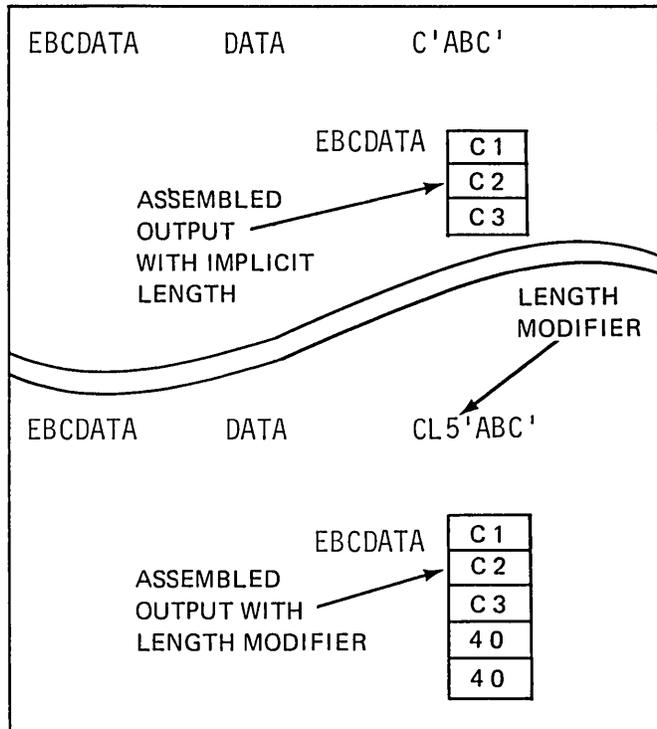


Figure 4-2. Length modifier

The length modifier is coded as Ln, where n = the number of bytes. In the lower example in Figure 4-2, a three-byte character string is placed in a five-byte field (length = L5), and the two extra bytes are padded with EBCDIC blanks (hex 40).

value nominal value of constant. The last part of the DATA statement operand is 'value'. When the DATA statement is assembled, the assembler initializes the number of data elements indicated (dup) of the desired type (type code) to the value coded in the 'value' part of the operand. Note that 'value' must always be coded, and for all data types other than address data (type code A), the value is enclosed in apostrophes.

The following examples illustrate the interaction of three parts of the DATA statement operand. (Length, since it is used with only two data types, will be ignored for the remainder of this discussion.)

```
DCON DATA F'0'
```

The example shown will define a one-word integer value, initialized to zero. The optional dup is not coded, so the length will default to the implicit length of the data type, which is one word for F type data.

```
CCON DATA 5C'A'
```

The example shows a data type of C (EBCDIC), and the duplication factor is 5. This statement would generate a five byte field of the EBCDIC representation of the character A (in hex, C1C1C1C1C1). The duplication factor applies to the data defined within the enclosing apostrophes of the value portion of the operand. If the DATA statement is written as follows;

```
CCON DATA 5C'ABC'
```

a fifteen-byte field would be defined, containing five repetitions of the ABC EBCDIC character string. Although the implicit length of an EBCDIC character is 1 byte, three characters are defined, so the duplication factor applies to the three-byte field.

The operand formats described do not apply when coding address (A-type) data constant. An A-type data constant is a single word in length, because it contains a Series/1 storage address.

```
ACON DATA A(FLC1)
```

The statement shown above will define a one-word constant at location ACON, containing the address of location FLC1. Note that the name of the location whose address you want in ACON is enclosed in parentheses, rather than apostrophes.

The DATA statement conforms to the rules for the Define Constant (DC) instructions in the BPPF Assembler. If you are not familiar with defining constants, it is recommended that you review the data definition section in the Series/1 Macro Assembler Language Reference (SC34-0317).

Here is a summary of the supported data types. The implicit length generated by the assembly of each different type code is indicated under Length.

1. Fixed Point Arithmetic Data

Type Code	Length
H	1 BYTE
F	2 BYTES (1 word)
D	4 BYTES (doubleword)

H, F, or D type codes define signed, fixed point values of the indicated length and are used in integer arithmetic operations.

2. Floating Point Arithmetic Data

Type Code	Length
E	4 BYTES
L	8 BYTES

E and L type codes generate standard or extended precision floating point constants, respectively. Floating point data is used in floating point arithmetic operations (Series/1 Floating Point hardware feature required).

3. Address Data Definition

Type Code	Length
A	2 BYTES (1 word)

The contents of the location defined will contain the address of a symbolic program location.

4. Hexadecimal/Binary

Type Code	Length
X	4 BITS
B	1 BIT

These allow definition of binary bit strings in storage, which are commonly used in logical operations and when using digital sensor I/O (DI/DO/PI). *Note:* Binary constants (type code B) cannot be defined if program preparation is being done using the online Program Preparation Facility, \$EDXASM.

5. Character Data

Type Code	Length
C	1 BYTE/CHARACTER

Defines EBCDIC characters in storage, for use with EBCDIC I/O devices (displays, printers).

## BUFFER STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706) "BUFFER."

The BUFFER statement provides a convenient way to define contiguous, named, storage areas in a program, for use in I/O operations, as work areas, etc. The BUFFER statement reserves space in storage, but does not initialize storage to a user-specified value. When the statement is assembled, the storage reserved is set to binary zeros, and will be zeros when the program containing the statement is initially loaded.

Figure 4-3 illustrates the format for the BUFFER statement, and shows what is generated in storage as a result. The label of the BUFFER statement is the symbolic name of the first data item. In storage this is preceded by two words of control information. The first word is called the INDEX, and may be symbolically referenced by the name you code in the INDEX= keyword operand of the BUFFER statement.

INDEX is used with SBIO and INTIME instructions to place data in sequential buffer positions automatically, and would not be coded unless the buffer being defined were intended for that purpose. See "Section 9. Timers" in this study guide for an example of the use of the INDEX operand.

The second word is the count, containing the buffer length you specified in the count operand. This count will be the number of words or bytes defined, depending on whether you coded BYTES for the item operand.

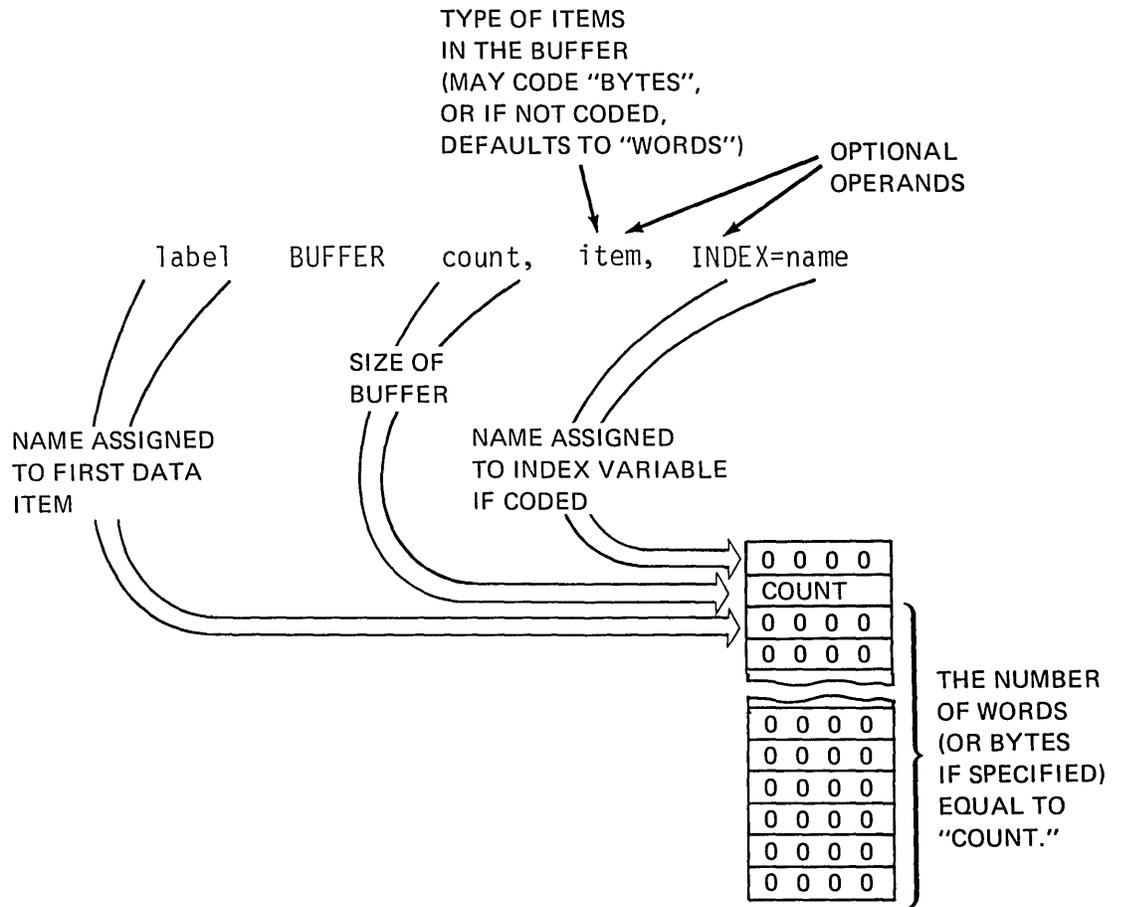


Figure 4-3. BUFFER statement

## TEXT STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "TEXT."

The TEXT statement is used to generate character buffers, and operates in conjunction with the terminal instructions READTEXT, PRINTTEXT, GETEDIT, and PUTEDIT. Figure 4-4 shows the format for the TEXT statement, and what is generated in storage.

**1** label TEXT 'message',LENGTH=,CODE=

**2** ENDMSG TEXT 'RUN ENDED',LENGTH=12,CODE=EBCDIC

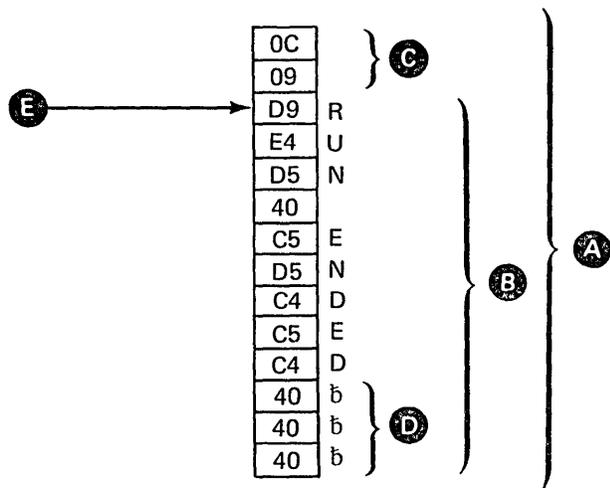


Figure 4-4. TEXT statement

In Figure 4-4, the TEXT statement format at **1** is shown coded at **2**. The message operand is the text 'RUN ENDED' in this example, but may be any character string you wish, up to 254 characters. The LENGTH= operand is coded as 12, indicating the total length of the text buffer. The CODE= operand is EBCDIC, which is also the default. The standard internal representation for character data is always EBCDIC. The system automatically converts the EBCDIC character strings to the code required by a particular terminal.

The CODE= operand could be coded ASCII. This is for special cases where you do not want the system to do any conversion from and to EBCDIC, but wish to transmit the exact code pattern in the buffer. An example is the graphics support, which drives a device employing an ASCII interface where certain ASCII characters perform graphics control functions.

The TEXT statement at **2** would generate the storage configuration shown just below it. The total storage utilized would be the 14 bytes shown by the brackets at **A**. The actual text buffer is defined within the brackets labeled **B**, encompassing 12 bytes (LENGTH=12). The data buffer is preceded by two bytes of control information, labeled **C**. The first byte defines the total length of the buffer (hex 0C), 12 bytes. The second byte is the length of this message, nine bytes, the total number of characters (including blank characters) in the 'message' operand. Unused character positions at the end of the buffer **D** are padded with blanks (EBCDIC for blank = hex '40'). The label of the TEXT statement points to the first byte of character data **E**.

For both input and output operations, the count (second byte at location **C**) cannot exceed the text buffer length (first byte at **C**). If you attempt to output a message that is larger than the buffer, or read a character string from a device that is longer than the buffer, the message will be truncated to fit within the defined buffer length.

The contents of the character buffer defined by a TEXT statement is not confined to the character string that was coded when it was assembled. Different messages may be moved into the buffer at different times during execution of a program. If data is moved into a TEXT buffer using the PUTEDIT command, the count byte is automatically adjusted to reflect the message length. When data is read from a terminal with a GETEDIT or a READTEXT command, the count reflects the number of input characters read. If a character string is moved into a TEXT buffer by any instructions other than these (i.e., MOVE), the count must be adjusted by the user before issuing a PRINTTEXT referencing that TEXT buffer.

This page intentionally left blank.

## DATA DEFINITION REVIEW EXERCISE – QUESTIONS

1. Match the type with the data representation
  - a. \_\_\_\_\_ Extended precision floating point      1. C
  - b. \_\_\_\_\_ Address      2. X
  - c. \_\_\_\_\_ Character      3. B
  - d. \_\_\_\_\_ Double word fixed point      4. F
  - e. \_\_\_\_\_ Half word fixed point      5. H
  - f. \_\_\_\_\_ Full word fixed point      6. D
  - g. \_\_\_\_\_ Binary      7. E
  - h. \_\_\_\_\_ Hexadecimal      8. L
  - i. \_\_\_\_\_ Standard precision floating point      9. A

2. Using the following instruction

```
MSG2 TEXT LENGTH=20
```

answer the following questions:

- a. How many characters could be stored in the text buffer defined?
  - b. How many words would be reserved?
  - c. How could you address the first character in the buffer?
3. How many words are reserved by the following instruction?

```
BUF3 BUFFER 16,BYTES
```

4. When coding a TEXT statement, if no 'message' is defined (LENGTH= only coded), the text buffer will be initialized to binary zeros.

True \_\_\_\_\_

False \_\_\_\_\_

## DATA DEFINITION REVIEW EXERCISE – ANSWERS

1.
  - a. 8
  - b. 9
  - c. 1
  - d. 6
  - e. 5
  - f. 4
  - g. 3
  - h. 2
  - i. 7
  
2.
  - a. 20 characters
  - b. 11 (20 bytes, one for each character, plus 2 bytes (one for length, one for count)).
  - c. By referencing the label MSG2
  
3. 10 words are reserved; 8 for the 16 data positions, and the two control words which precede the data.
  
4. False. Undefined text buffer locations are initialized to EBCDIC blanks (hex 40).

## Section 5. Data Manipulation

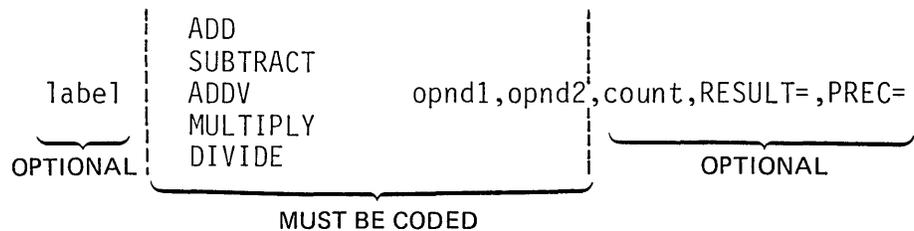
**OBJECTIVES:** After successful completion of this topic, the student should be able to:

1. Understand the Event Driven Executive arithmetic instructions which operate on signed integer variables
2. List the Event Driven Executive arithmetic instructions which operate on floating point data
3. Use the Event Driven Executive data movement instructions to:
  - a. Replace the contents of one variable with that of another
  - b. Replace the contents of a variable with the address of another
  - c. Replace the contents of a data field with the contents of another data field
4. Determine the result of executing any of the Event Driven Executive logical instructions, given the values represented by operand1 and operand2

### INTEGER ARITHMETIC

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Data Manipulation", "Data Representation", "Mixed Precision Operations", "ADD", "ADDV", "SUBTRACT", "MULTIPLY", "DIVIDE."

Figure 5-1 shows the basic format of instructions that operate on integer arithmetic variables.



**Figure 5-1.** Integer arithmetic instruction format

Data flow is from opnd2, to opnd1; in the ADD or SUBTRACT instructions, the data represented by opnd2 is added to or subtracted from the data represented by opnd1, and the result of the operation replaces the contents of the location specified by opnd1.

In the MULTIPLY or DIVIDE instructions, the data in opnd1 is multiplied or divided by the data in opnd2, and the product or quotient replaces the contents of opnd1 (for DIVIDE; the remainder is stored in the task code word, and will be overlaid by the next DIVIDE, I/O or floating point operation).

## Optional Operands

The optional operands (count, RESULT=, and PREC=) allow the application programmer to control the number of variables involved in the operation, where the result of the operation should be placed, and to specify the size of the variables (word, doubleword) used. The following examples illustrate how the optional operands affect instruction execution. An ADD operation is used as an example, but the principles also apply for SUBTRACT, MULTIPLY, and DIVIDE.

```
EXAMPLE1  ADD  VAL1,CONWORD
```

This first example uses no optional operands, and is the most basic form. The word at location CONWORD will be added to the word at location VAL1. The results of the operation will replace the contents of VAL1. Both VAL1 and CONWORD are assumed to be single precision (word-length) signed integer variables, because word-length is the default when no other precision is specified.

```
EXAMPLE1  ADD  VAL1,CONWORD,5
```

The count operand is coded as a 5. The count operand references opnd1, and specifies how many variables, beginning at the location specified in opnd1, the contents of opnd2 should be added to. In the example shown, the word at location CONWORD would be added to the word (still the default precision) at location VAL1, to the word at location VAL1+2 (two bytes = one word), at VAL1+4, and so on through location VAL1+8. Each of the words in the five word field beginning at location VAL1 would be increased by the value of the contents of location CONWORD.

```
EXAMPLE1  ADD  VAL1,CONWORD,5,RESULT=RFIELD
```

Without changing anything else, the keyword operand RESULT= has now been added. This statement will execute the same way as did the previous example except that the results of the operation will be placed in a five-word field beginning at location RFIELD. The five words beginning at location VAL1 will remain unchanged.

The only remaining optional operand is the keyword PREC=, which allows the programmer to specify the precision of the opnd1 and opnd2 variables. Again using our example, if the field of data beginning at location VAL1 were double precision integers, and we wanted to add a single precision integer at location CONWORD to each of them, PREC=D would be coded.

```
EXAMPLE1  ADD  VAL1,CONWORD,5,RESULT=RFIELD,PREC=D
```

The results (double precision integers) would be placed in a ten word field beginning at location RFIELD, leaving the original contents of VAL1 undisturbed.

The D in PREC=D signifies that opnd1 is double-precision. DD would have indicated that both opnd1 and opnd2 were double precision. See "Mixed Precision Operations" in the Language Reference manual for allowable opnd1/opnd2 precision combinations.

Thus far, the count optional operand referred to opnd1 only. The vector addition capability is an exception to that rule. The ADDV statement adds the corresponding components of two vectors together, and therefore the count operand specifies the number of components in both vectors (opnd1 and opnd2).

## FLOATING POINT ARITHMETIC

The format for Floating Point instructions is similar to that for the arithmetic instructions handling integer variables, except that the optional count operand is not allowed. Floating point operations involve the two discrete values represented by opnd1 and opnd2 only; neither may be vectors.

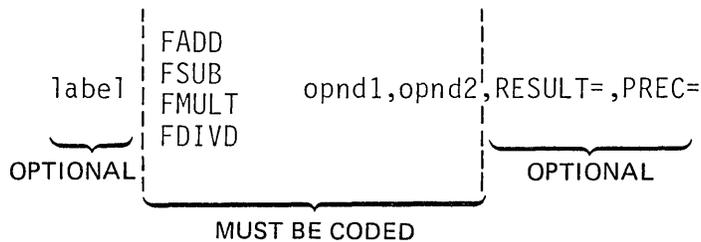


Figure 5-2. Floating point arithmetic instruction format

The floating point instructions are not software simulations of floating point hardware; the Series/1 Floating Point hardware feature must be installed to utilize the floating point capability.

Support for both standard and extended precision variables (PREC= operand), and all precision combinations are allowed.

For an example of the use of floating point instructions, see Example 6 in the Language Reference, SC34-1706.

## DATA MOVEMENT INSTRUCTIONS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "MOVE", "MOVEA."

The MOVE statement has the following format:

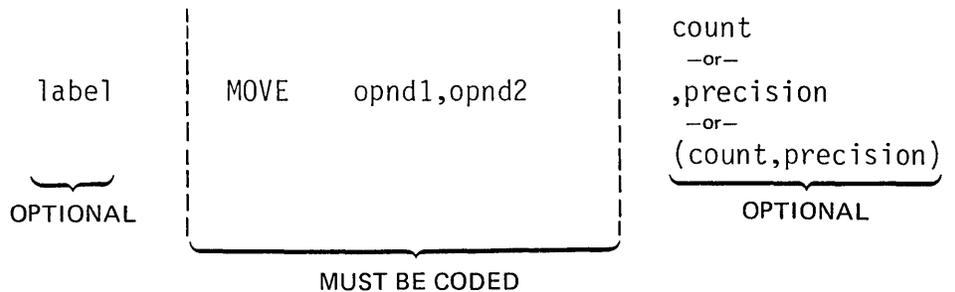


Figure 5-3. MOVE instruction format

Unlike the integer and floating point arithmetic instructions, the RESULT= optional keyword operand is not used; the data specified by opnd1 is always replaced by that represented by opnd2. The following statement,

```
MOVE    OLDDATA,NEWDATA
```

would replace the word (default precision) at location OLDDATA with the word at NEWDATA.

The same operation, coded with the count operand=3,

```
MOVE    OLDDATA,NEWDATA,3
```

would move the three words starting at location NEWDATA into the three words starting at location OLDDATA.

For MOVE statements, precision is indicated by the keywords BYTE, WORD (default) or DWORD (doubleword). If count is not coded (default count = 1), then precision is coded by itself. If count is coded, precision is included as a sublist element in the count operand.

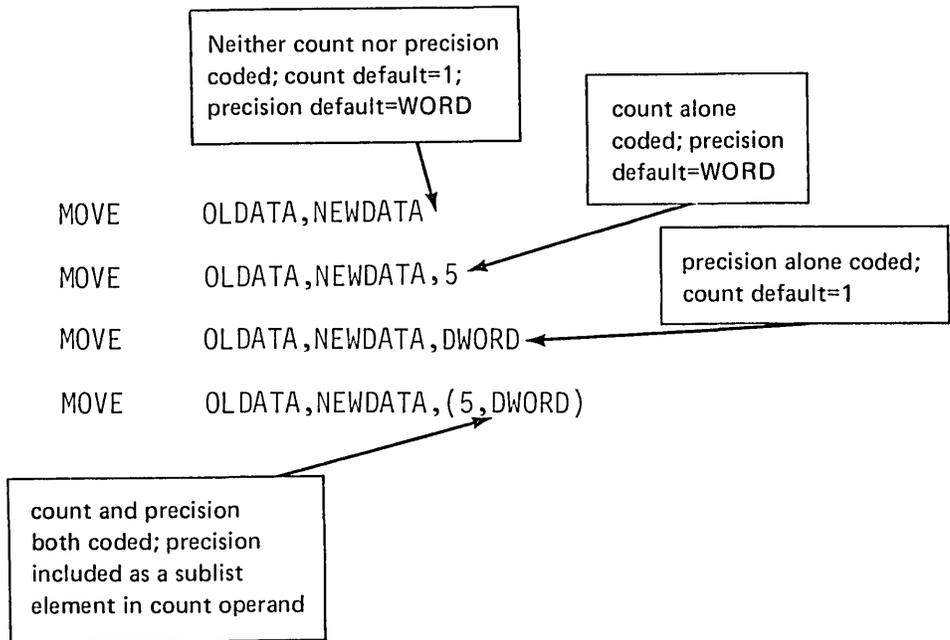


Figure 5-4. MOVE optional operands

Move operations move data from a field of specified length, to a field of equal length, so count applies to both opnd1 and opnd2.

The following examples illustrate the MOVE instruction optional operand variations. Each of the instructions is logically equivalent, moving four bytes of data from opnd2 to opnd1.

```

MOVE OLDDATA,NEWDATA,(4,BYTE)
MOVE OLDDATA,NEWDATA,2
MOVE OLDDATA,NEWDATA,(2,WORD)
MOVE OLDDATA,NEWDATA,DWORD
MOVE OLDDATA,NEWDATA,(1,DWORD)

```

The MOVEA instruction moves the *address* of the location specified in opnd2 into the location specified by opnd1.

```

MOVEA DATADRS,DATA

```

In the example shown, the address of location DATA replaces the contents of location DATADRS. No optional operands are allowed with the MOVEA statement, because:

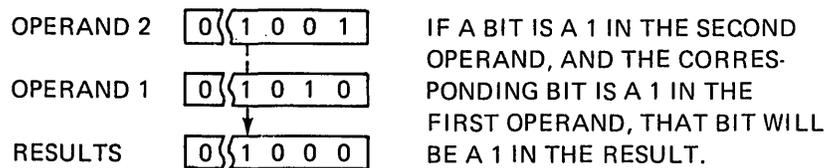
- a. opnd1 is always the target of the move, so RESULT= is not valid
- b. the data being moved is a Series/1 storage address, which is, by definition, word-length; precision is therefore always WORD (no PREC= coded)
- c. only a single address at a time is moved, so count is always = 1, and is therefore not coded.

## LOGICAL INSTRUCTIONS

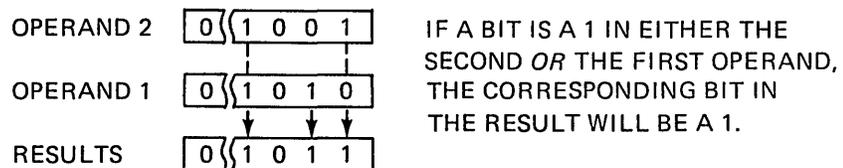
READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "AND", "IOR", "EOR", "SHIFTL", "SHIFTR."

The logical instructions AND (AND), OR (IOR), and exclusive OR (EOR) operate upon selected bits within a bit field. Opnd2 operates on opnd1 in the manner summarized in Figure 5-5.

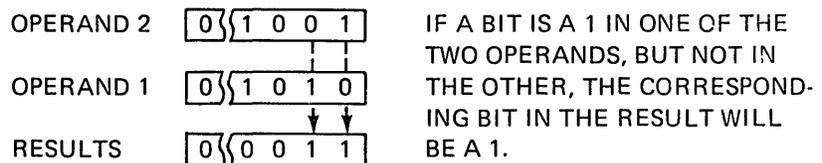
### AND (AND)



### OR (IOR)



### Exclusive OR (EOR)



**NOTE:** RESULTS OF AND, IOR, EOR OPERATIONS WILL REPLACE THE CONTENTS OF OPERAND 1, OR WILL BE PLACED IN THE LOCATION SPECIFIED IN THE RESULTS= OPERAND, IF IT IS CODED.

Figure 5-5. AND/OR/exclusive OR

The instruction format for AND, IOR, and EOR is shown in Figure 5-6. As with MOVE operations, precision may be BYTE, WORD (default), or DWORD. The precision applies to opnd1, opnd2, and to RESULT=, if coded. The count optional operand applies to opnd1 and RESULT= only; count for opnd2 is always =1.

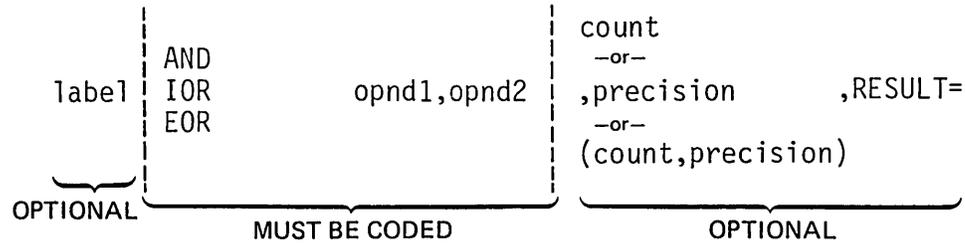


Figure 5-6. Logical instruction format

If RESULT= is coded, the contents of opnd1 are unchanged by the operation. The following illustrates the use of the optional operands.

```
AND    XDATA, ZDATA
```

Since count, precision, and RESULT= are not coded, count defaults to 1, precision defaults to WORD, and the contents of XDATA will be replaced by the word-length bit-field resulting from the AND of the 16 bits in the word at ZDATA with the 16 bits in the word at XDATA.

```
AND    XDATA, ZDATA, 3
```

The contents of XDATA, XDATA+2, and XDATA+4 will be replaced by the results of the AND of the 16 bits in the word at ZDATA with each of the 16 bits beginning at XDATA. Note that the same word at ZDATA is consecutively ANDed with the three-word bit field beginning at location XDATA. The precision (default=WORD) determined how many bits at a time to AND (opnd2 size), and the count operand how many consecutive groups of bits of that size to perform the operation against.

```
AND    XDATA, ZDATA, (3, BYTE)
```

The above is the same as the operation shown before, except that the 8 bits specified in opnd2 (BYTE precision) are successively ANDed against the three 8-bit groups in opnd1, beginning with the byte at location XDATA.

AND XDATA,ZDATA,(3,BYTE),RESULT=YDATA

When the statement above is executed, the three bytes, beginning at location YDATA, will be replaced by the results of the AND of the byte at location ZDATA with the three bytes in XDATA, XDATA+1, and XDATA+2.

Event Driven Executive logical instruction capability also includes logical shift operations, for both shift left (SHIFTL) and shift right (SHIFTR). (See Figure 5-7.) Logical shifts, like the other logical instructions, operate on bit-fields (bit-strings).

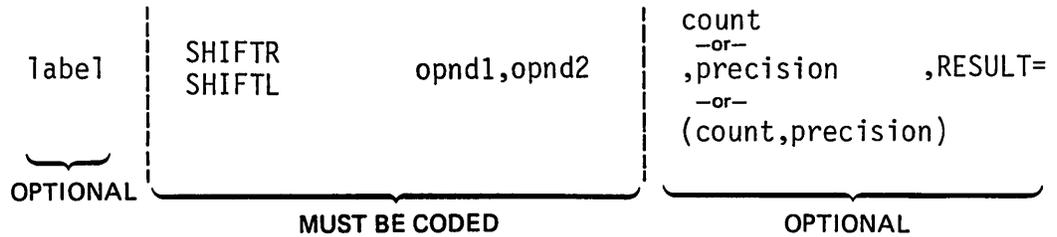
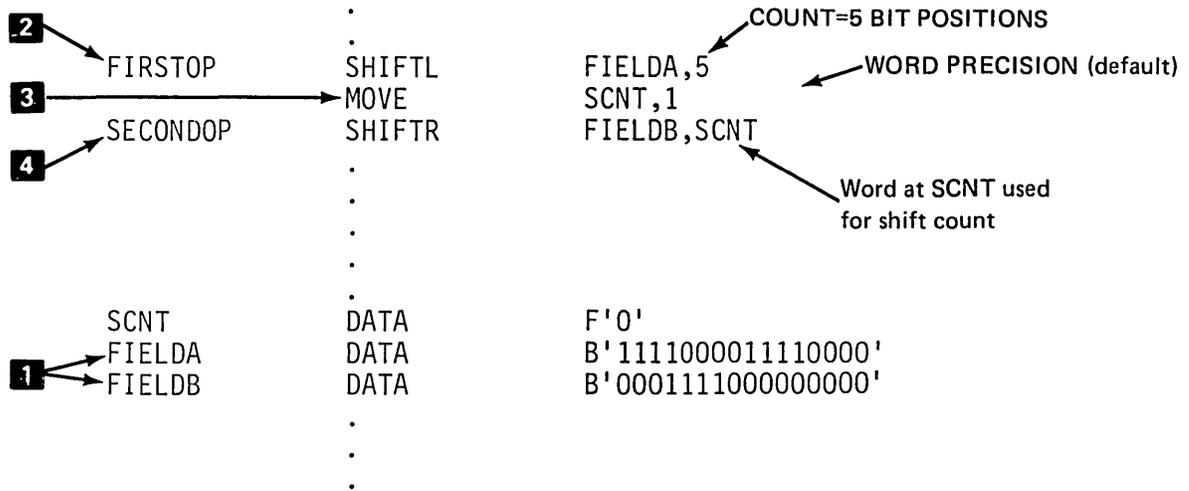


Figure 5-7. Shift instruction format

In shift operations, opnd2 is coded as an absolute value or as a variable name. The absolute value, or the contents of the variable, contains the shift count (the number of bit positions, to the right or left, that the contents of the bit field which begins at location opnd1, should be shifted).

The optional operands have the same meaning, and are coded in the same way, as for AND, IOR, and EOR (note that if opnd2 is a variable name, that variable has the same precision (BYTE,WORD,DWORD) as the variable opnd1).

A SHIFTL instruction shifts bits out of the high-order (most significant) position of a bit field, and fills vacated low-order (least significant) bit positions with zeroes. Similarly, SHIFTR shifts bits out of the low-order position, and zero-fills vacated high-order positions. Figure 5-8 illustrates the operation of both SHIFTL and SHIFTR.



**1** Before execution of the Shift Left at FIRSTOP, the contents of FIELDA and FIELDDB are exactly as coded

**2** After execution of the Shift Left at FIRSTOP;

FIELDA = 0001 1110 0000 0000

1111 0 Shifted out of high order position

zeros filled in vacated bit positions

**3** After execution of the MOVE operation, location SCNT=1

**4** After execution of Shift Right at SECONDDOP,

FIELDA = 0001 1110 0000 0000, unchanged,

and FIELDDB = 0000 1111 0000 0000

zero fills vacated position

0 shifted out of low order bit position

Figure 5-8. Shift operation

## DATA MANIPULATION REVIEW EXERCISE – QUESTIONS

1. Fill in the value for X, Y, and Z after execution of each of the instructions below. In each case, assume that before execution, X=20, Y=30, and Z=0.

a.                   ·  
                      ·  
                      ADD    X,Y  
                      ·  
                      ·

Answers:   X= \_\_\_\_\_   Y= \_\_\_\_\_   Z= \_\_\_\_\_

b.                   ·  
                      ·  
                      ADD    X,Y,RESULT=Z  
                      ·  
                      ·

Answers:   X= \_\_\_\_\_   Y= \_\_\_\_\_   Z= \_\_\_\_\_

c.                   ·  
                      ·  
                      ADD    X,50  
                      ·  
                      ·

Answers:   X= \_\_\_\_\_   Y= \_\_\_\_\_   Z= \_\_\_\_\_

2. Analyze the two arithmetic operations below, and explain how they would differ when executed.

a.   ·  
   ·  
   ·   ADD    X,Y,2  
   ·  
   ·  
   ·

b.   ·  
   ·  
   ·   ADDV    X,Y,2  
   ·  
   ·  
   ·

ANSWER: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

3. Analyze the two data movement operations below, and explain how they would differ when executed.

a. MOVE X,Y                      b. MOVEA X,Y

ANSWER: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

4. Below is a coding example using all five logical instructions. Each instruction uses the "RESULT=" optional keyword operand to place the result in a different location (opnd1 is undisturbed). Fill in (in binary) what the "RESULT=" locations would be after execution of the coding example.

```
.  
. AND XDATA,ZDATA,BYTE,RESULT=ANDRSLT  
. IOR XDATA,ZDATA,BYTE,RESULT=IORRSLT  
. EOR XDATA,ZDATA,BYTE,RESULT=EORRSLT  
. SHIFTR ZDATA,7,BYTE,RESULT=RITERSLT  
. SHIFTL XDATA,3,BYTE,RESULT=LEFTRSLT  
. .  
. .  
. .  
. .  
. .  
XDATA DATA B'11010010'  
ZDATA DATA B'10011001'  
. .
```

ANSWERS:

After execution,

- a. ANDRSLT= B'
- b. IORRSLT= B'
- c. EORRSLT= B'
- d. RITERSLT=B'
- e. LEFTRSLT=B'

## DATA MANIPULATION REVIEW EXERCISE – ANSWERS

1. a. X50      Y30      Z0  
b. X20      Y30      Z50  
c. X70      Y30      Z0
2. Example a. (ADD operation) would add the contents of storage location "Y" to storage location "X" and to storage location "X+2". The "count" operand (2) applies to opnd1 only.  
Example b. (ADDV operation) would add the contents of storage location "Y" to storage location "X", and the contents of storage location "Y+2" to the contents of storage location "X+2". The "count" operand of the ADDV instruction applies to both opnd1 and opnd2 (also for MOVE).
3. Example a. (MOVE operation) would replace the contents of storage location "X" with the contents of storage location "Y" (move Y to X). Example b. (MOVEA operation) would replace the contents of storage location "X" with the *address* of the storage location "Y" (move the address of Y to X).
4. a. ANDRSLT=B'10010000'  
b. IORRSLT=B'11011011'  
c. EORRSLT=B'01001011'  
d. RITERSLT=B'00000001'  
e. LEFTRSLT=B'10010000'

## Section 6. Queue Processing

**OBJECTIVE:** After completing this topic, the student should be able to:

1. Define an empty or a full queue
2. Add entries to a queue
3. Retrieve the oldest entry from a queue
4. Retrieve the newest entry from a queue

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Queue Processing."

The queuing instructions discussed in this section are used to define queues and access entries in queues. The size of a queue (the number of entries it can hold) is specified by the user. A queue entry is one word in length. The contents of this word may comprise the queue entry in its entirety, or as in the examples used in this section, may be the address of a larger data area (buffer).

A useful example of queue definition and processing is buffer pool management. If several tasks within an application program have the possibility of performing I/O operations, a queue of I/O buffers (buffer pool) can be established. Using the queue processing instructions, a task requiring an I/O buffer obtains it from the pool, and, when the I/O has completed, returns it to the pool. No physical movement of the buffer is involved; the queue entry that is acquired and returned is actually the address of the buffer in storage.

Another example of the use of queue processing is a "data spooling" operation, where multiple units of data are placed in a direct access data set, with the record numbers of the first record of each unit stored as a data element (entry) in a queue for later processing. In this instance, the single-word queue entry is the queued data item itself, rather than a pointer to a storage location or buffer.

## DEFINEQ

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "DEFINEQ."

For this discussion, a *queue* is the system mechanism and control blocks necessary to logically connect and manage a chain of queue entries. Figure 6-1 shows the format of the DEFINEQ statement, which is used to establish a queue.

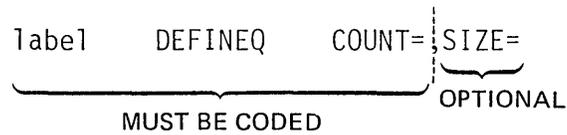


Figure 6-1. DEFINEQ format

The label of the DEFINEQ statement is a required field. It is the symbolic name of the queue, and will be used by queue processing instructions to access the queue. The COUNT= keyword operand (coded as an integer value) determines the number of Queue Control Elements (QCEs) and therefore, the possible number of associated buffer pool elements the queue may reference. QCEs are three-word system control blocks, which are logically (contain address pointers) chained together in active or free QCE chains. QCEs in the active chain include data entries; free chain QCEs contain no data entries, and are connected to other free QCEs.

In addition to QCEs, the DEFINEQ statement also generates a single Queue Control Block (QCB). The QCB is three words long, and the first word is assigned the label of the DEFINEQ statement. The QCB contains address pointers to the active and free chains of QCEs. When an entry is added to a queue, the QCB address pointers are adjusted to remove a QCE from the free chain and attach it to the active chain.

SIZE= is an optional keyword operand. It may be coded to cause the generation of a pool of data buffers associated with the queue being defined. The number of such buffers will equal that specified in the COUNT= operand. The size of each buffer (in bytes) is specified by the integer value coded in the SIZE= operand. If SIZE= is not coded, no buffer pool will be generated, and all QCEs will initially be defined to be in the free chain (empty queue). If SIZE= is coded, all QCEs will be in the active chain (full queue), and the entry in each active QCE will point to one of the buffers in the buffer pool.

In Figure 6-2, the SIZE= operand is not coded, so an empty queue is defined (all QCEs in free chain). In figure 6-2, and in the rest of the illustrations in this section, QCEs in the free chain are shown as shaded.

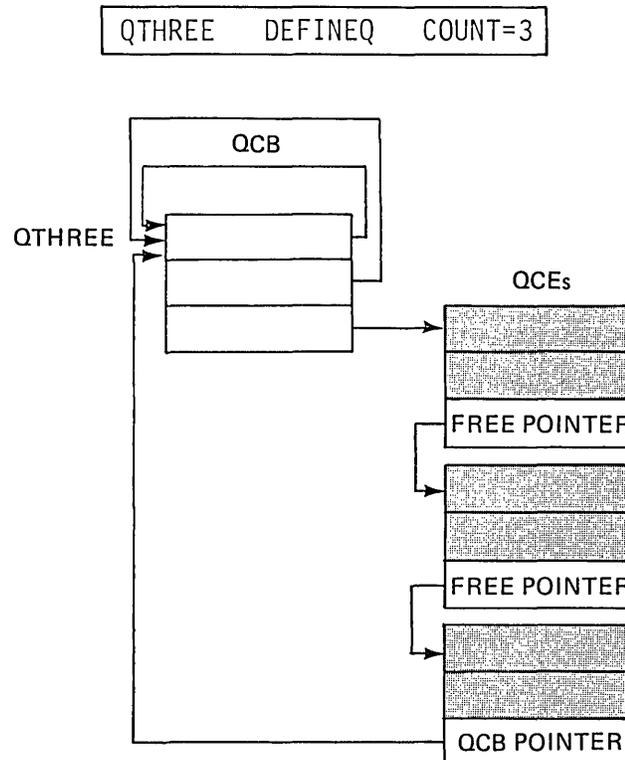


Figure 6-2. Empty queue

No entries are in the queue, but there is space (free QCEs) available for the addition of three entries.

In Figure 6-3, a full queue (all QCEs in active chain, with queue entries pointing to buffer pool elements) is defined. Each buffer pool element is four bytes in length (SIZE=4). No more entries may be added to this queue, as all QCEs are already active.

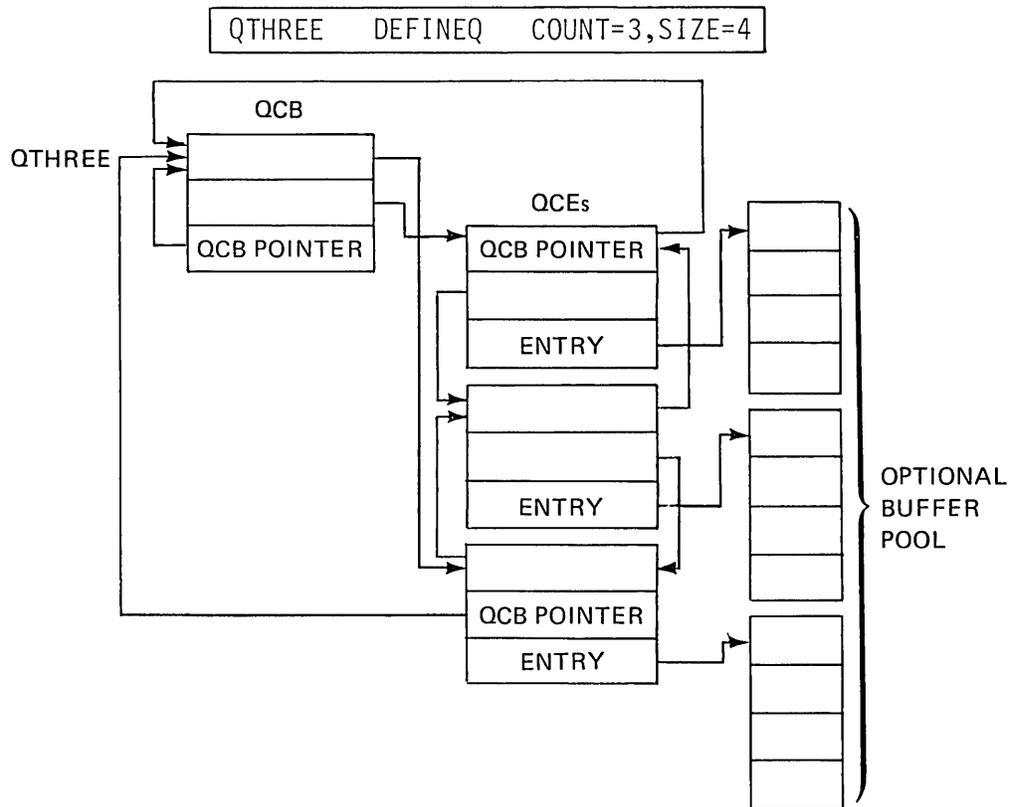


Figure 6-3. Full queue

### LASTQ/FIRSTQ/NEXTQ

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "NEXTQ", "FIRSTQ", "LASTQ."

The queue processing instructions allow the user to add (NEXTQ) or retrieve (LASTQ, FIRSTQ) entries in a queue defined by the DEFINEQ statement. The format for all three queue processing instructions is similar:

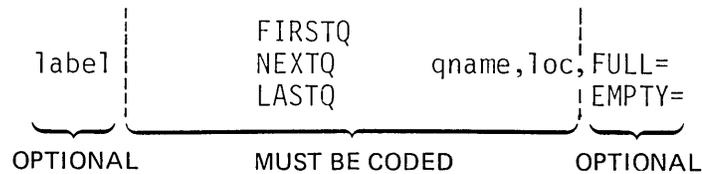


Figure 6-4. Queue processing instruction format

FIRSTQ and LASTQ are used to retrieve entries from a queue; NEXTQ places an entry in a queue. The label of a DEFINEQ statement is coded as *qname*, specifying which queue is being accessed.

The loc operand is the label of a one-word storage location. This word will be set to the contents of the entry being retrieved from the queue by a FIRSTQ or LASTQ instruction. Before executing a NEXTQ instruction, the user must ensure that this word contains the entry (data item, such as a record number; or address of a buffer pool element) being added to the queue.

The EMPTY= keyword operand is coded as the label of the instruction that will receive control if the queue referenced by a FIRSTQ or LASTQ instruction has no active entries. FULL= performs the same function for the NEXTQ instruction in the event there is no room in the queue to add an entry. If EMPTY= or FULL= is not coded, and the queue is erroneously empty or full, execution will continue with the instruction following the FIRSTQ/LASTQ or NEXTQ. A +1 will be returned in the task code word (taskname), and may be checked by the user.

Entries are placed in a queue one at a time. Therefore, queue entries differ in their relative age, as some are queued before others. Both FIRSTQ and LASTQ retrieve entries from a queue, but they differ in the age of the entries they retrieve.

LASTQ retrieves the last, and therefore the most recently entered, entry in a queue. This is often called "Last In, First Out", or LIFO queue processing. It is also referred to as stack processing.

This page intentionally left blank.

## QUEUE PROCESSING REVIEW EXERCISE—QUESTIONS

1. Including all control blocks, how many bytes of storage will be reserved by the DEFINEQ statement below?

QEXAMP      DEFINEQ      COUNT=5,SIZE=256

Answer: \_\_\_\_\_ bytes

2. What instruction would you execute to:
  - a. Retrieve the oldest entry in a queue \_\_\_\_\_
  - b. Add an entry to a queue \_\_\_\_\_
  - c. Retrieve the most recent entry in a queue \_\_\_\_\_
3. Figure 6-4 shows the format for the Queue Processing instructions. What optional keyword operand would be used to branch to a user routine:
  - a. When you attempt to retrieve a queue entry and there are no active entries \_\_\_\_\_
  - b. When you attempt to add an entry to a full queue \_\_\_\_\_

## QUEUE PROCESSING REVIEW EXERCISE—ANSWERS

1.     6                    QCB 3 words, 2 bytes/word  
      30                    QCEs 5 QCEs, 3 words, 2 bytes/word  
      1280                BUFFERS 5 of 256 bytes each  
      1316 bytes
2.    a. FIRSTQ  
       b. NEXTQ  
       c. LASTQ
3.    a. EMPTY =  
       b. FULL =

## Section 7. Program Control

OBJECTIVES: Upon successful completion of this topic, the student should be able to:

1. Explain the use and execution of subroutines in an application program
2. Incorporate Assembler language routines in an Event Driven Executive program

### SUBROUTINES

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "SUBROUT", "CALL", "RETURN."

In many programs, there are certain functions that are required repeatedly at different points in the program's execution. Examples might include conversion of data from one code to another or a particular sequence of arithmetic calculations.

Rather than code the sequence of instructions that perform the desired function each time the program needs that function, the function is coded once, and defined as a subroutine. The subroutine can then be entered and executed from as many different points in the application program as required.

### SUBROUT STATEMENT

Subroutines are defined using the SUBROUT statement whose format is shown in Figure 7-1.

$$\underbrace{\text{label}} \quad \underbrace{\text{SUBROUT} \quad \text{name}} \quad \underbrace{, \text{par1}, \dots, \text{par5}}$$
  
OPTIONAL      MUST BE CODED      OPTIONAL

Figure 7-1. SUBROUT format

The *name* operand is coded with the symbolic name of the subroutine and will be referenced by other instructions. The *label* field is optional, and should not be confused with the subroutine name specified in the name operand.

Par1 through par5 are names of parameters that may be passed to the subroutine when it is entered.

## CALL STATEMENT

The format of the CALL statement is shown in Figure 7-2. The CALL is used to enter a subroutine defined in a SUBROUT statement.

label CALL name, par1, ..., par5  
OPTIONAL MUST BE CODED OPTIONAL

Figure 7-2. CALL format

The name operand is coded with the symbolic name specified in the name operand of the SUBROUT statement defining the subroutine you wish to execute. Par1 through par5 may be coded as single precision integer values, as the symbolic names (labels) of single precision integer values, or as the addresses of program variables or data areas.

## PASSING SUBROUTINE PARAMETERS

Figure 7-3 illustrates basic subroutine operation. Note that the CALL at location START is a call to CALC, not to SUB1, the label on the SUBROUT statement. The last executable statement in this and every subroutine is a RETURN. The RETURN instruction provides the linkage back to the calling task, where execution resumes at the instruction following the CALL. Subroutines execute as part of, and at the same priority as, the calling task. Subroutines are not re-entrant, so if a subroutine is called from multiple tasks, ENQ and DEQ should be used to ensure serial execution.

SUBEXAMP	PROGRAM	START
START	CALL	CALC
	.	
	.	
	.	
	.	
	.	
	PROGSTOP	
INTEGERA	DATA	F'10'
INTEGERB	DATA	F'15'
SUM	DATA	F'0'
SUB1	SUBROUT	CALC
	ADD	INTEGERA,INTERGERB,RESULT=SUM
	.	
	.	
	.	
	.	
ENDIT	RETURN	
	ENDPROG	
	END	

Figure 7-3. Subroutine operation

The subroutine CALC in Figure 7-3 adds two integer values together and stores the result at location SUM. Since CALC is part of program SUBEXAMP, all labels within the program are known to the subroutine, and may be referenced by instructions within the subroutine. In this example, location SUM would contain 25 after the subroutine has been executed.

When a subroutine uses specific labels in the program, the data that the subroutine will operate on must be moved into the storage addresses represented by those labels before the subroutine is called. The same result can be achieved more easily by using the parameter passing capability. Parameters may be actual values (integer numbers), or may take the form of pointers to data that the subroutine will be using.

In figure 7-4, the SUBROUT statement at location SUB1 specifies two parameters, XVAL and YVAL. The names used to define parameters in SUBROUT statements must be unique throughout the program (cannot appear in the label field of any statement). They are positional symbolic references to parameters that are passed in the CALL statement.

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	

Figure 7-4. Integer parameters

In the first CALL (location START), the first parameter is the single precision integer value 50. This corresponds to the first parameter defined in the SUBROUT statement, XVAL, as does program location SUM1 to the second parameter definition YVAL. When the ADD instruction at location A1 executes as a result of this call, the value 50 will be substituted when XVAL is referenced, and location SUM1 will be used in place of YVAL. Location SUM1 will be set to 60, the sum of INTEGERA and 50.

The second CALL at C2 will result in 70 being put in location SUM2, the sum of SUM1 and INTEGERA. Notice that although INTEGERA is used by the subroutine, it need not be passed as a parameter, since it does not change from CALL to CALL.

Up to this point, the parameters illustrated have been restricted to single precision integer values. By passing an address of a data area as a parameter, and utilizing the software registers (#1, #2) within the subroutine, any data area or data array may be accessed.

In Figure 7-5, the address of the data area SUMAREA is passed as the first parameter of the CALL (label is enclosed in parentheses to specify *address* rather than *content of address*). When the subroutine executes the address is loaded into software register #1. The results of the ADD operations are moved into SUMAREA using the contents of #1 as a base address. After execution, SUMAREA will contain 50, and SUMAREA+2 will contain 25.

SUBEXAMP	PROGRAM	START
START	CALL	CALC,(SUMAREA),40,INTERGERB
	.	
	.	
	.	
	.	
	.	
	PROGSTOP	
SUMAREA	EQU	*
	DATA	2F'0'
INTEGERA	DATA	F'10'
INTEGERB	DATA	F'15'
	SUBROUT	CALC,ADDRSLT,XVAL,YVAL
	MOVE	#1,ADDRSLT
	ADD	INTEGERA,XVAL,RESULT=S1
	MOVE	(0,#1),S1
	ADD	INTEGERA,YVAL,RESULT=S1
	MOVE	(2,#1),S1
	RETURN	
S1	DATA	F'0'
	ENDPROG	
	END	

Figure 7-5. Address parameter

When employing this technique, you should keep in mind that the software registers used by subroutines are those associated with the calling task, and therefore, the subroutine may be required to save them on entry and restore them to their original values before returning.

*Note:* If a subroutine is assembled as a separate module for later link editing, the subroutine name must be declared in an ENTRY statement.

## USER STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "USER."

At some time you may require a function not provided by the Event Driven Executive. Such functions can be coded in Series/1 assembler language (assuming that you have the appropriate assembler language background) and included in an Event Driven Executive program as a *user exit routine*. The USER statement provides the linkage between the Event Driven Executive code and the Series/1 assembler language routine.

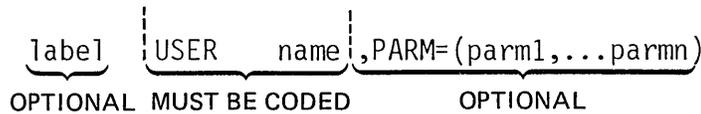


Figure 7-6. USER format

The name operand is coded as the label of the entry point (label of first executable instruction) of the assembler language routine. The PARM= keyword operand is coded as a list of parameters, with each parameter as a sublist element.

When executing Event Driven Executive code, the user is limited to the two software registers, #1 and #2. In Series/1 assembler language, the hardware registers are available. Since the Event Driven Executive system uses these hardware registers also, certain conventions must be observed when execution switches from Event Driven Executive code to Series/1 assembler language and back again. First, hardware register 2 (R2) is always pointing to the Task Control Block of the task currently in execution, and must not be disturbed. Second, hardware register 1 (R1) is used by the system to provide linkage to and from Event Driven Executive instructions. When a user exit routine is entered (branched to by a USER instruction), R1 is pointing to the next instruction following the USER statement, where Event Driven Executive language execution will resume when the assembler language routine completes. If parameters are passed by the USER statement (PARM= coded), R1 will be pointing to the location containing the first parameter. Before exiting from the assembler language code, the user must increment R1 past all parameters so that it points to the Event Driven Executive instruction following the USER statement.

The program in Figure 7-7 includes the user exit routine S1CODE. When the USER statement at location START is executed, a branch to label S1CODE is performed.

Two parameters are coded in the PARM= parameter list of the USER statement. As with the CALL statement, each parameter is one word in length, consisting of an integer value or the address of a program location. Upon entry to S1CODE, R1 is pointing to the first parameter, which contains the integer value 9. The MVW at location S1CODE moves the integer value to location FRSTPARM.

The second parameter is the address of program location XVAL. Using the indirect addressing capability, R1 is again used to move the parameter into the subroutine.

USERXAMP	PROGRAM	START
START	USER	S1CODE, PARM=(9, XVAL)
A1	ADD	P3, FIVEB
	.	
	.	
	.	
	.	
	.	
	.	
	.	
	PROGSTOP	
XVAL	DATA	F'0'
P3	DATA	F'0'
FIVEB	DATA	F'0'
	.	
	.	
	.	
S1CODE	MVW	(R1,0), FRSTPARM
GET2	MVW	(R1,2)*, SECDPARM
	.	
	.	
	.	
	.	
	.	
	.	
	.	
	.	
UPDATE	ABI	4, R1
OUT	B	RETURN
FRSTPARM	DC	F'0'
SECDPARM	DC	F'0'
	ENDPROG	
	END	

Figure 7-7. User exit routine

To go back to Event Driven Executive code from a user exit routine, you must branch to label RETURN (B RETURN), as shown at location OUT. The system routine RETURN expects to find R1 pointing to the next Event Driven Executive instruction following the USER statement. The ABI instruction, at location UPDATE, increments R1 past the two words in the parameter list, so that it points to the ADD instruction at location A1.

User exit routines can only be assembled by \$S1ASM (Series/1 macro assembler) or host macro assemblers. To incorporate a user exit routine into a program prepared using the Program Preparation Facility, the routine must be first assembled using \$S1ASM or the host assembler, and the resulting object module linked to the Event Driven Executive main program using \$LINK. The user exit routine entry point should be defined in an ENTRY statement, and the same entry point must be coded in an EXTRN statement in the main program with which the routine will be linked.

**PROGRAM CONTROL REVIEW EXERCISE – QUESTIONS**

1. What statement is coded to transfer control to a subroutine written in Event Driven Executive language?

Answer: \_\_\_\_\_

2. Event Driven Executive subroutines begin with a \_\_\_\_\_ statement, and the last statement to be executed must be a \_\_\_\_\_ statement.

3. Why can't user exit routines be assembled using \$EDXASM?

Answer: \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

4. How does executing a subroutine differ from executing a secondary task?

Answer: \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

5. What statement is used to transfer control to a user exit routine?

Answer: \_\_\_\_\_

6. How can you pass more than five parameters to an Event Driven Executive subroutine?

Answer: \_\_\_\_\_

\_\_\_\_\_  
\_\_\_\_\_

## PROGRAM CONTROL REVIEW EXERCISE – ANSWERS

1. CALL
2. SUBROUT, RETURN
3. User exit routines are written in Series/1 assembler language, and the \$EDXASM assembler can assemble Event Driven Executive language only. User exit routines are assembled using the Series/1 Macro Assembler \$S1ASM, or a host macro assembler.
4. A secondary task executes concurrently with the attaching task, and may be run at a different priority. A subroutine executes on the priority of the calling task, and "in-line" with the execution of the calling task.
5. USER
6. Use one of the five parameters to pass the address of a data area to the subroutine. The data area can contain as many additional parameters as required.

## Section 8. Program Sequencing

**OBJECTIVES:** Upon successful completion of this topic, the student should be able to:

1. explain the operation and use of
  - a. unconditional GOTO
  - b. indirect GOTO
  - c. computed GOTO
2. define an IF/THEN/ELSE/ENDIF structure
3. define a DO/ENDDO structure
4. explain the use of relational statements with IF and DO statements
5. combine IF, DO, and GOTO statements in logical code sequences

### GOTO STATEMENT

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Program Sequencing", "GOTO."

Almost all programs have multiple execution paths. A different sequence of execution may be necessary because of the characteristics of the input data, the results of a calculation, or the occurrence of an exception or error condition. One of the Event Driven Executive instructions providing the means to transfer control to an alternate section of code is the GOTO statement.

Figure 8-1 is an example of the most basic form of the GOTO statement. This is an unconditional GOTO, used to branch around a section of non-executable code (e.g., data definitions) that are imbedded within the executable code.

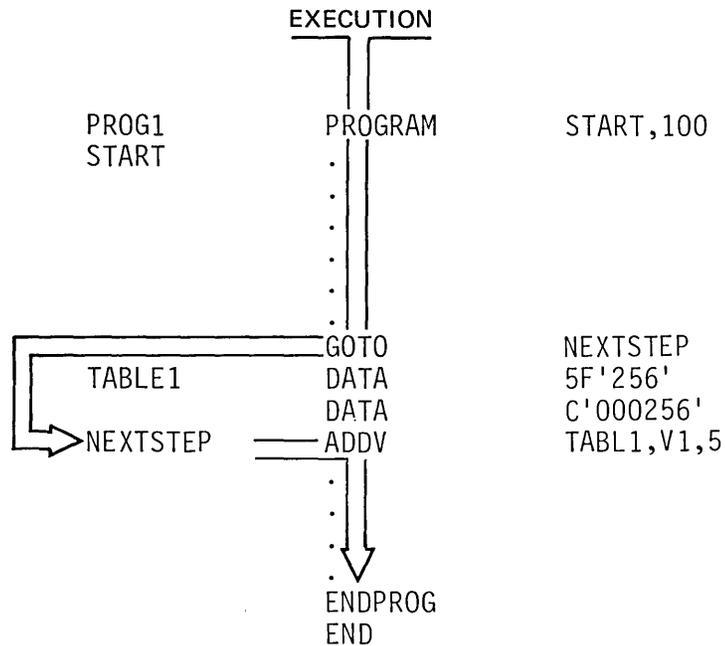


Figure 8-1. Unconditional GOTO

Control is transferred from the GOTO statement to the statement at location NEXTSTEP, skipping over the two DATA statements which start at TABLE1.

Figure 8-2 illustrates another form of GOTO. In this example, the operand is enclosed in parentheses, indicating an indirect GOTO. During PROG1 program execution, but prior to executing the GOTO instruction, the address of the desired "branch to" location (Address of NEXTSTEP) is moved **1** into location BRNCHADR **2**.

BRNCHADR is the name defined within parentheses in the operand of the GOTO statement **3**. When the GOTO is executed, control is transferred to the instruction at NEXTSTEP **4**, indirectly through the contents of BRNCHADR.

The indirect GOTO can serve as an unconditional branch to any label in a program, as long as the address of the desired destination is first moved into the indirect address location coded as the operand of the GOTO.

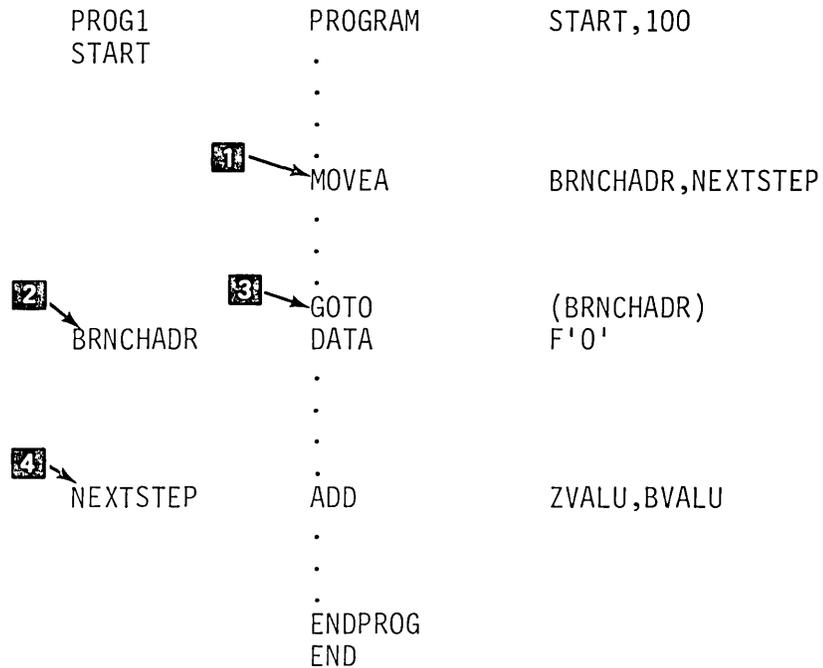


Figure 8-2. Indirect GOTO

A third form of GOTO statement is the computed GOTO, whose format is shown in Figure 8-3.

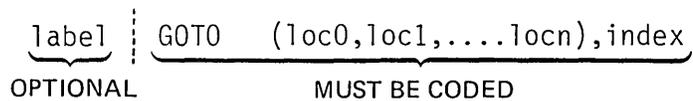


Figure 8-3. Computed GOTO format

In the first operand, loc0 through locn are the symbolic addresses of instructions to which control may be transferred. The second operand is an index variable. The address to which control is transferred is determined by the value of the index variable.

The first address (loc0) in the list of addresses which form the first operand is the address to which you want control transferred if the index variable exceeds the extents of list loc1–locn.

The next address in the list, loc1, will get control if the index variable is equal to 1, loc2 if the index variable is equal to 2, etc.

Figure 8-4 illustrates the operation of a computed GOTO with an index variable outside the range of the list. The index variable is VAL1 and is set to zero by the MOVE statement at location "START". Zero is outside the range of loc1–locn (NDX1, NDX2 in this case), and the computed GOTO transfers control to the address at loc0 (ERROR).



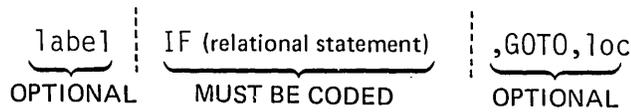


Figure 8-5. IF Format

The first operand is a relational statement, and all IF statements must have at least one relational statement. A relational statement expresses a comparative relationship between two variables, or between a variable and an explicit value. An IF may be coded to include a GOTO (second operand) and a specified location (third operand). For instance, (Figure 8-6);

```

.
.
TEST1  IF  (A,EQ,B),GOTO,STEP3
.
.

```

Figure 8-6. IF/GOTO example

This statement may be interpreted as "Transfer control to location STEP3 if the value in location A is equal to the value in location B." If A is not equal to B, execution will continue with the instruction following the IF. The "IF with GOTO" is the simplest form of IF that can be coded. IF statements may also take the form of *structures*, in which entire code sequences may be executed or skipped, depending on whether the relationship expressed in the relational statement is true or not. The basic IF structure is illustrated in Figure 8-7.

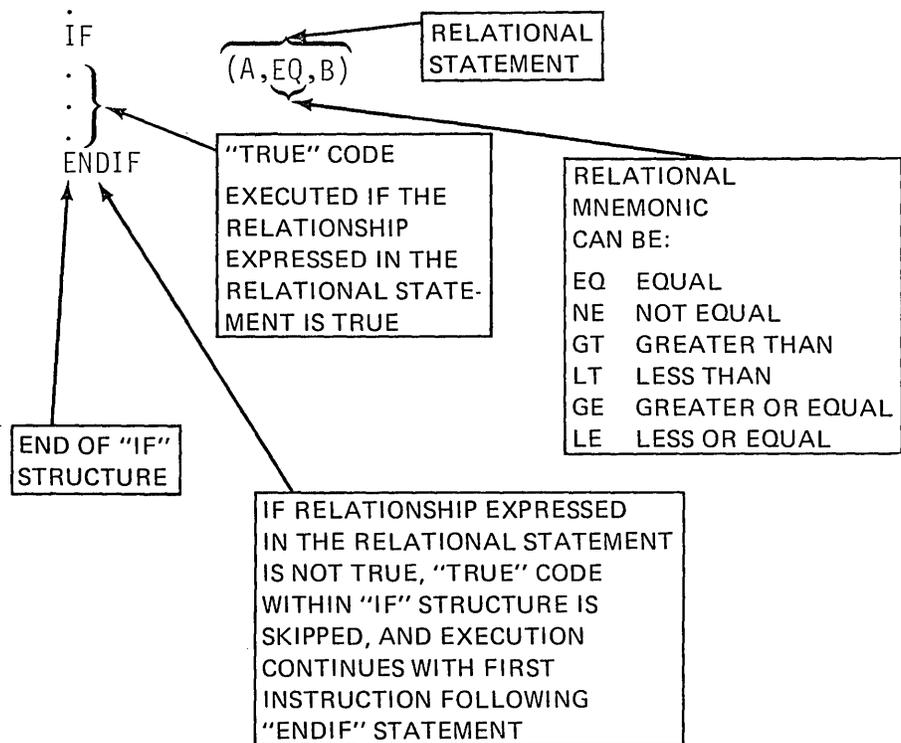


Figure 8-7. IF structure

All IF structures must end with an ENDIF statement, except when using GOTO. In the example, the code between the IF statement and the ENDIF will be executed if the relationship expressed in the statement is true (A is equal to B). If the relationship is not true, the *true* code will be bypassed, and execution will continue with the statement following the ENDIF.

In Figure 8-8, one more statement is added to the IF structure. The ELSE statement starts the false code; these instructions will be executed if the relationship expressed in the statement is not true, bypassing the "true" code. True code begins following the IF in an IF structure, and ends with the ENDIF if no ELSE statement is coded (Figure 8-7), or ends with an ELSE statement if one is used (Figure 8-8).

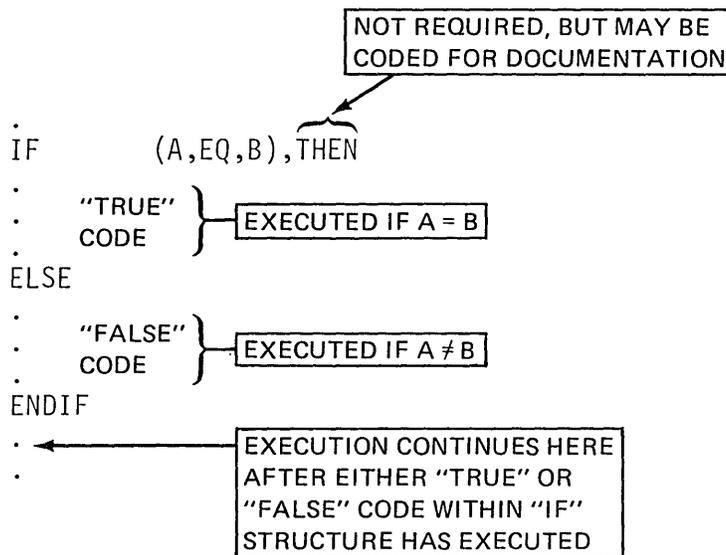


Figure 8-8. IF/THEN/ELSE

False code begins with an ELSE statement, and ends with the ENDIF, which defines the end of that IF structure.

## Relational Conjunctions

As you found in the reading assignment, IF structures can be very complex. Figure 8-9 is an example of a structure using logical conjunctions and nesting. A logical conjunction forms a logical link between two or more relational statements. A *nested* IF structure is one that appears within the true or false code of a previous IF structure.

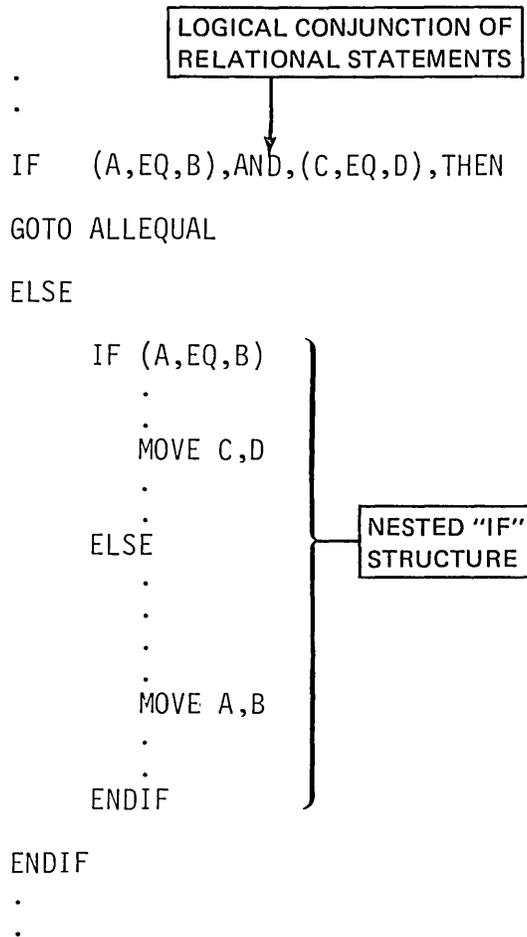


Figure 8-9. Complex IF structure

A transfer to ALLEQUAL will take place only if both 1) A=B and 2) C=D. The false code is another IF structure, nested within the first, with its own true and false sections. Notice that each IF structure is ended with its own ENDIF statement.

## DO STATEMENT

READING ASSIGNMENT: IBM Sereis/1 Event Driven Executive Language Reference (SC34-1706), "DO", "ENDDO."

The DO instruction alters the sequence of program execution by causing repetitive execution of the same section of code. The DO statement establishes the start of a DO loop, and the end of the loop is defined by an ENDDO statement. The code that is repeatedly executed is the instruction or instructions that are coded between the DO and ENDDO statements.

One form of the DO statement is illustrated in Figure 8-10. The count operand is an integer value, or the label of a storage location containing an integer value, indicating the number of times you want to execute the loop.

$\underbrace{\text{label}}$      $\underbrace{\text{DO count, TIMES, INDEX=}}$   
 OPTIONAL    MUST BE CODED    OPTIONAL

Figure 8-10.

TIMES has no function other than documentation, and does not have to be coded. The INDEX= keyword operand may be coded as the label of a word of storage. Before the DO loop is executed for the first time, the storage location is reset to zero. Then, before execution of the first instruction following the DO statement, and with every succeeding pass, 1 is added to the storage location. In the event that a branch out of the loop is done before the count has gone to zero, the location specified in the INDEX= operand can be checked to see how many executions occurred.

Figure 8-11 is a flowchart representing the execution sequence of the DO count, TIMES form of DO loop. (If the INDEX= operand is not coded, the top two blocks would not apply.)

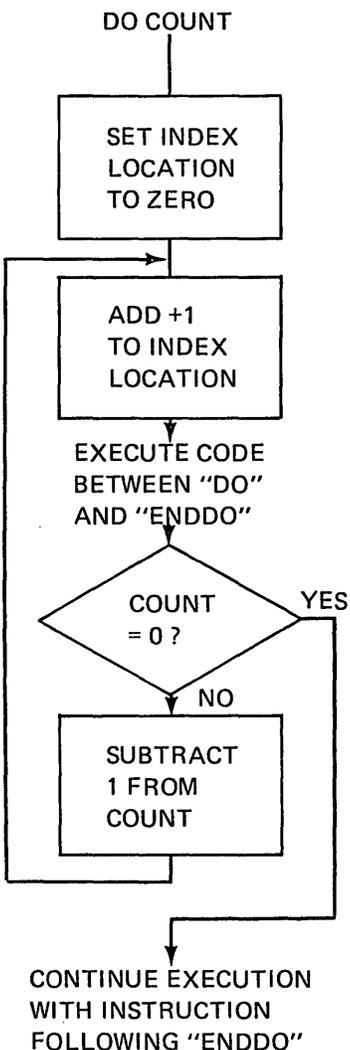


Figure 8-11. "DO count" operation

Notice that a post-execution escape mechanism is used (trailing decision loop). The count is not checked for zero until the loop has completed the first execution. Therefore, if count is initially zero, one execution would still occur.

There are two other forms of the DO statement, both employing relational statements. DO WHILE will repetitively execute the instructions within the loop while the relationship expressed remains true. DO UNTIL will keep on executing the loop until the relationship expressed in the relational statement becomes true. The format for these two instructions is illustrated in Figure 8-12.



Figure 8-12. WHILE/UNTIL format

The relational statements are coded the same way as those used with the IF statement, and like the IF, two or more relational statements may be formed into a statement string, using the logical conjunctions AND and OR.

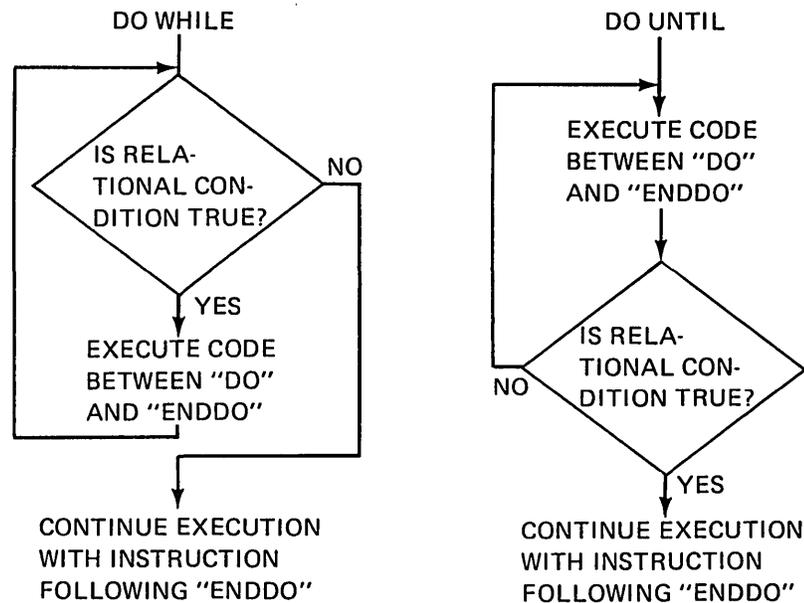


Figure 8-13. WHILE/UNTIL operation

Figure 8-13 illustrates the execution sequence of DO WHILE and DO UNTIL. DO WHILE has a pre-execution (leading decision loop) escape mechanism. The relational condition is checked before the first execution and, if not true, no execution takes place. DO UNTIL, like DO count, does not check until completing the first execution of the loop. Even if the relational condition is true, one execution will occur.



## PROGRAM SEQUENCING REVIEW EXERCISE – QUESTIONS

Using the coding example below, answer the questions which follow.

```

      .
      .
IF1ST      IF      (A,NE,B)
IF2ND      IF      (A,GT,B),THEN
            SUB      A,B
            MOVE     VAL1,A
ELSE2ND    ELSE
            SUB      B,A
            MOVE     VAL1,B
END2ND     ENDIF
ELSE1ST    ELSE
            GOTO     EXIT4
END1ST     ENDIF
COMPGO     GOTO     (ERR,EXIT1,EXIT2,EXIT3),VAL1
      .
      .
```

1. Assuming that A=5, and B=3, the next statement to be executed after execution of the code in the example is at location
  - a. ERR
  - b. EXIT1
  - c. EXIT2
  - d. EXIT3
  - e. EXIT4
2. Assuming that A=22, and B=23, the next statement to be executed after execution of the code in the example is at location
  - a. ERR
  - b. EXIT1
  - c. EXIT2
  - d. EXIT3
  - e. EXIT4
3. Assuming A=0, and B=-5, the next statement to be executed after execution of the code in the example is at location
  - a. ERR
  - b. EXIT1
  - c. EXIT2
  - d. EXIT3
  - e. EXIT4

4. The "true" code for the IF structure beginning at location IF1ST consists of
- a. the code starting at IF2ND and ending at ELSE2ND
  - b. the code starting at IF2ND and ending at END2ND
  - c. the code starting at IF2ND and ending at END1ST
  - d. none of the above
5. If control is transferred to location EXIT4, then the following is true;
- a. VAL1=4
  - b. A is greater than B
  - c. B is greater than A
  - d. A and B are equal
  - e. none of the above
6. How many times will the DO loop below execute?

```
.  
. DO          17,TIMES,INDEX=TWO  
.   
.   
.   
.   
.   
.   
.   
.   
. ENDDO  
.   
.
```

Answer: \_\_\_\_\_

7. Using the coding example below, pick the correct statement from the list of statements which follow

```

      .
      .
D01   DO UNTIL,(X,EQ,Y),OR,(Y,GT,X)
D02   DO WHILE,(X,EQ,Y)
D03   DO UNTIL,(X,NE,Y)
      ADD Y,1
      ENDDO
ENDDO3
ENDDO2
ENDDO1
      .
      .
      .
```

Assume when execution begins,  $X=Y$ .

- a. All three DO loops will execute one time.
- b. The first two DO loops will execute once, but the innermost DO loop (DO3 to ENDDO3) will not be executed.
- c. None of the DO loops will execute, because X is equal to Y when the first DO statement is encountered (DO1).
- d. Question is not valid, because DO loops cannot be nested.

## PROGRAM SEQUENCING REVIEW EXERCISE – ANSWERS

1. The correct answer is choice c. A is not equal to B, so the "true" code following the IF at location IF1ST will be executed. A is greater than B, so the "true" code of the nested IF at IF2ND is executed. VAL1 is set to 2, the result of the SUBTRACT operation. Execution continues at location COMPGO, skipping the "false" code of the nested IF and the first IF. VAL1, the index variable of the computed GOTO at location COMPGO was set to 2 by the statements in the preceding IF structure, so control is transferred to location EXIT2.
2. The correct answer is choice b. A is not equal to B, so the "true" code of IF1ST is executed. A is not greater than B, so the "false" code of the nested IF (ELSE2ND to END2ND) is executed, and the difference between A and B is placed in VAL1 (VAL1=1). The computed GOTO at COMPGO will transfer control to location EXIT1.
3. The correct answer is choice a. Execution proceeds exactly as in the answer to question 2 above ( $A \neq B, A < B$ ), but the difference between A and B is 5. When the computed GOTO at COMPGO is executed, the index variable, VAL1, contains a value which exceeds the range of the list, and therefore control is transferred to location ERR.
4. Choice b is the correct answer. "True" code is everything between the IF and the ELSE statement/or the IF and the ENDIF if ELSE is not coded.
5. Choice d is correct. If A and B are equal, the relational statement in the IF at location IF1ST is false, and the "false" code is executed. The "false" code is the unconditional GOTO at location EXIT4.
6. The DO loop will execute 17 times. The index variable, TWO, will be set to zero before the first execution of the DO loop, and assuming that the code within the DO loop does not contain any GOTO statements, the loop will execute 17 times, and the index variable TWO will contain 17 after the DO loop is exited.
7. The correct answer is choice a. Although X and Y are equal at the time the first DO statement is executed (DO1), the relational condition associated with a DO UNTIL statement is not checked until after the first execution of the DO loop.

The second DO loop (DO2) starts with a DO WHILE statement. The DO WHILE checks for the relational condition before executing for the first time, but since the condition is true, execution drops to the second nested DO loop at DO3.

The innermost DO loop is another DO UNTIL, this time with a "NOT EQUAL" relational mnemonic. The ADD operation within the loop makes the two variables, X and Y not equal, thereby satisfying the exit condition for DO3, the innermost loop.

The exit condition for the second loop, DO2 (first nested loop) is also satisfied, because it is supposed to execute only as long as X is equal to Y, which is no longer true.

The first loop will also exit, because although X is not equal to Y, which is the relational condition specified in the first part of the relational statement, Y is greater than X, which is specified in the second part of the relational statement, and the two parts are joined by the OR conjunction. All three loops will therefore exit after a single execution.

*Note:* The relational statement used with the DO at location DO1 could have been coded as:

```
DO1          DO          UNTIL,(Y,GE,X)
```

and would have executed with the same effect as the form used in the example.



This page intentionally left blank.



## Section 9. Timers

**OBJECTIVES:** After completing this topic, the student should be able to:

1. Use the GETTIME instruction to access the time-of-day and date from an application program
2. Use the INTIME instruction to measure time intervals
3. Cause user-defined delays in task execution by using the STIMER instruction along with the "WAIT on timer" capability

If you have the hardware timer feature installed on your Series/1 4955 Processor, or your processor is a 4952 (has self-contained timer), you can include support in your Event Driven Executive supervisor, which provides several time/timing functions that may be used by application programs. In addition to maintaining a time-of-day clock, the system also provides a time interval (elapsed time) clock, and has the capability to suspend task execution (go into wait state) for specified lengths of time.

### GETTIME INSTRUCTION

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "GETTIME."

The time-of-day (TOD) clock is maintained in hours, minutes, and seconds. At initial program load (IPL), the clock is all zeros and begins running. It may be set to actual clock time using the \$T operator command, and will maintain clock time from that point on.

The GETTIME instruction is used to move the TOD values into a user program. The GETTIME format is;

$$\underbrace{\text{label}} \quad \underbrace{\text{GETTIME}} \quad \underbrace{\text{loc, DATE=}}$$
  
OPTIONAL    MUST BE CODED    OPTIONAL

Figure 9-1. GETTIME format

The hours, minutes, and seconds are maintained by the system in three storage words in the supervisor. The user must define a three word storage area in the application program issuing the GETTIME, into which the hours, minutes, and seconds can be moved. The loc operand is coded as the label of the first position of the three word user-defined area.

The \$T operator command also allows you to enter the date in the form of month-day-year or day-month-year (depending on how the DATEFMT= keyword operand of the SYSTEM statement was coded during system generation). If the DATE= keyword operand is coded DATE=YES, the GETTIME instruction will transfer the date as well as the time into the application program. Three words are also required for the date, and these must be contiguous with and following the three word area defined to hold the time.

Each of the six words in the TOD and date locations are direct binary equivalents of the information they represent. For instance, the third word of TOD information (loc+4) is seconds, and when it reaches 59, the next increment resets it to zero, and the minutes word is increased by 1 (loc+2). Hours is increased by 1 when 60 minutes have elapsed, days by 1 at midnight, etc. By using GETTIME, an application program can time stamp reports, transactions, or any system event in which information as to the actual time of occurrence is useful.

## INTIME INSTRUCTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "INTIME."

Some applications need to measure elapsed time: how long it takes for a certain code sequence, task or program to execute, or how much time has passed between the occurrences of events. These time intervals may be very short, and therefore, cannot be accurately measured using TOD values, whose resolution is only to the nearest second.

In addition to the TOD clock, the system maintains a relative time clock. It consists of a double precision (two-word) integer, which is initialized to zero at system IPL. Every millisecond thereafter, this value is incremented by 1, and at any given instant, therefore, contains the elapsed time in milliseconds since the system IPL. (A double-precision integer will contain a count of milliseconds comprising approximately 49 days elapsed time, before rolling over to zero and starting again.)

The INTIME instruction is used to read the relative time clock value into a user program. The format for the INTIME statement is shown in Figure 9-2.

label	INTIME	reltime,loc	,INDEX
OPTIONAL	MUST BE CODED		OPTIONAL

Figure 9-2. INTIME format

The reltime operand is coded as the label of a user-defined double-precision integer variable into which the relative time value will be moved. The loc operand is coded as the label of a user-defined single precision integer, which will be set to the number of milliseconds that have passed since an INTIME instruction, referencing this reltime location, was executed in this program. (A single-precision integer will hold approximately 65 seconds elapsed time in milliseconds, before rolling over to zero and starting again.)

The INDEX keyword, if coded, indicates that automatic indexing is to be used in conjunction with a BUFFER statement. If INDEX is coded, the loc operand must be the label of a BUFFER statement, instead of a single-word integer. When automatic indexing is used, repetitive executions of an INTIME instruction result in the storing of successive elapsed time values in successive buffer positions. The use of INTIME with automatic indexing is illustrated at the end of this section, along with the other timer instructions.

## STIMER INSTRUCTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "STIMER."

Every task has a software timer associated with it. This timer will time out after a user-specified number of milliseconds has elapsed (60 seconds or 60,000 milliseconds maximum). The desired time interval is set and the timer started by the STIMER instruction, whose format is illustrated in Figure 9-3.

label
| STIMER    count |
| WAIT

OPTIONAL
MUST BE CODED
OPTIONAL

Figure 9-3. STIMER format

The count operand is coded either as the number of milliseconds you want to elapse before the timer expires, or as the label of a word of storage containing the desired number of milliseconds. If the WAIT keyword is coded, the task will go into the wait state until the specified time interval has passed. Execution will resume with the instruction following the STIMER.

The WAIT does not have to be coded as part of the STIMER instruction, but may appear later as an explicit WAIT on the keyword operand TIMER. This acts in the same manner as a wait on an event, the event being expiration of the time delay. Using this method, the timer is started, and execution continues with the instruction following STIMER. When the WAIT on TIMER is encountered, the WAIT will fall through if the time interval has already passed, or the task will go into a wait state for the amount of time remaining.

## TIMING FUNCTIONS – CODING EXAMPLE

Figure 9-4 is a program that exercises all of the timing functions previously discussed in this section. The first instruction in the program is GETTIME at location STARTIME. It will place the TOD values for hours, minutes, and seconds into the three words defined at location STARTED.

The DO loop starting at DOSTART and ending at DOEND will execute three times. Each time, the INTIME instruction at location I1 will place the time elapsed since IPL in the double precision integer at SINCEIPL, and will put the time that has elapsed since the last INTIME execution in the next successive buffer location of the buffer defined at TIMEBUF. Both values are in milliseconds.

The STIMER instruction at location S1 causes a 5 second delay (5000 milliseconds = 5 seconds) in each execution of the DO loop. After the third delay, the DO loop exits, and the STIMER at location S2 executes. This starts a 10 second timer running but, since the WAIT operand is not coded, execution continues.

TIMESTEST	PROGRAM	STARTIME
STARTIME	GETTIME	STARTED
DOSTART	DO	3,TIMES
I1	INTIME	SINCEIPL,TIMEBUF,INDEX
S1	STIMER	5000,WAIT
DOEND	ENDDO	
S2	STIMER	10000
I2	INTIME	SINCEIPL,LASTIME
ENDWAIT	WAIT	TIMER
G2	GETTIME	STOPPED,DATE=YES
	PROGSTOP	
STARTED	DATA	3F'0'
SINCEIPL	DATA	2F'0'
TIMEBUF	BUFFER	3
LASTIME	DATA	F'0'
STOPPED	DATA	6F'0'
	ENDPROG	
	END	

Figure 9-4. Timing functions

The INTIME instruction at I2 places the elapsed time since IPL into SINCEIPL again, and puts the elapsed time since a previous INTIME instruction referencing SINCEIPL was executed into the single precision integer at LASTIME (INDEX not coded). The WAIT at ENDWAIT puts the program in a wait state, until the expiration of the 10 second time delay that was started by the STIMER at S2.

When the 10 seconds are up, the GETTIME at G2 executes, and the program ends. This time DATE=YES is coded, so a six-word area is defined at location STOPPED. Hours, minutes, and seconds will be placed in the first three words, and month, day, and year in the next three.

When using INTIME to time events where a few milliseconds difference is critical, keep in mind that the time values retrieved by your program represent the time that the INTIME instruction is executed. If the task issuing the INTIME is of a lower priority than other tasks active in the system at the same time, a delay in execution of the INTIME may result, and will be reflected in the clock value retrieved.

**This page intentionally left blank.**

## TIMERS REVIEW EXERCISE – QUESTIONS

All of the questions in this Review Exercise refer to the program in Figure 9-4. For simplicity, assume that no time is used to execute instructions, no other tasks are running in the system, and system overhead is zero.

At the time that the program begins execution, the date has been set at January 1st, 1979, and it is exactly 5 p.m. (1700 hours). The system IPL was at exactly 4 p.m.

1. What will be in the three words beginning at location STARTED after execution of the GETTIME at location STARTIME?

Answer:    STARTED    \_\_\_\_\_

              STARTED+2 \_\_\_\_\_

              STARTED+4 \_\_\_\_\_

2. What will be the values in the double precision integer at SINCEIPL and the buffer at TIMEBUF after the first execution of the INTIME instruction at I1?

Answer:    SINCEIPL    \_\_\_\_\_

              TIMEBUF    \_\_\_\_\_

              TIMEBUF+2 \_\_\_\_\_

              TIMEBUF+4 \_\_\_\_\_

3. After the second execution?

Answer:    SINCEIPL    \_\_\_\_\_

              TIMEBUF    \_\_\_\_\_

              TIMEBUF+2 \_\_\_\_\_

              TIMEBUF+4 \_\_\_\_\_

4. After the third execution?

Answer:    SINCEIPL    \_\_\_\_\_

              TIMEBUF    \_\_\_\_\_

              TIMEBUF+2 \_\_\_\_\_

              TIMEBUF+4 \_\_\_\_\_

5. What will be in SINCEIPL and in LASTIME after execution of the INTIME instruction at location I2?

Answer:    SINCEIPL    \_\_\_\_\_

              LASTIME    \_\_\_\_\_

6. What will be in the six words beginning at location STOPPED after execution of the GETTIME at location G2?

Answer:    STOPPED    \_\_\_\_\_

              STOPPED+2 \_\_\_\_\_

              STOPPED+4 \_\_\_\_\_

              STOPPED+6 \_\_\_\_\_

              STOPPED+8 \_\_\_\_\_

              STOPPED+10 \_\_\_\_\_

**TIMERS REVIEW EXERCISE – ANSWERS**

1.   STARTED        17  
       STARTED+2    0  
       STARTED+4    0

The TOD clock is kept using military time, on a 24 hour-a-day basis. Five p.m. is therefore 17 hours, 0 minutes, and 0 seconds.

2.   SINCEIPL       3,600,000  
       TIMEBUF       0  
       TIMEBUF+2    0  
       TIMEBUF+4    0

If the system IPL was at 4 o'clock, and it is now 5 o'clock, the relative time clock has been running for one hour, or 3,600,000 milliseconds. (1 hr x 60 minutes x 60 seconds x 100 milliseconds/second). The first word in TIMEBUF is zero, because the elapsed time from the last time an INTIME instruction referencing SINCEIPL was executed is zero; this is the first time the INTIME has executed.

3.   SINCEIPL       3,605,000  
       TIMEBUF       0  
       TIMEBUF+2    5,000  
       TIMEBUF+4    0

The second time through, the 5 second delay at S1 has occurred. Total elapsed time since IPL has increased by 5,000 milliseconds (SINCEIPL), and the time elapsed since the first INTIME execution, also 5000 milliseconds, is automatically indexed into TIMEBUF+2.

4.   SINCEIPL       3,610,000  
       TIMEBUF       0  
       TIMEBUF+2    5,000  
       TIMEBUF+4    5,000

A second 5 second delay has occurred, increasing SINCEIPL by another 5000 milliseconds, and placing 5000 milliseconds in the third buffer position, TIMEBUF+4.

5.   SINCEIPL       3,615,000  
       LASTIME       5,000

Before exiting the DO loop, an additional 5 second delay occurred, adding another 5000 milliseconds to SINCEIPL. Because the INTIME instruction references the same "reltime" operand as the last INTIME execution (SINCEIPL), LASTIME is set to 5000 milliseconds. If the INTIME at I2 had a different "reltime" operand, it would be treated as a first execution, and LASTIME would indicate zero elapsed time.

6.	STOPPED	<u>17</u>	5 p.m.
	STOPPED+2	<u>0</u>	0 minutes
	STOPPED+4	<u>25</u>	25 seconds
	STOPPED+6	<u>1</u>	January
	STOPPED+8	<u>1</u>	1st
	STOPPED+10	<u>79</u>	1979

Fifteen seconds in the DO loop, plus the 10 second delay at S2 have elapsed.

This page intentionally left blank.

## Section 10. Disk/Diskette I/O

**OBJECTIVES:** Upon successful completion of this topic the student should be able to:

1. Understand the logical layout of disk, diskette and tape
2. Define data sets in a PROGRAM statement
3. Read records using the READ statement
4. Write records using the WRITE statement
5. Use NOTE and POINT to access and set the next record indicator
6. Pass data set definitions to programs loaded from a terminal or from another program
7. Pass data set definitions to an overlay program from the program loading the overlay

### DEVICES SUPPORTED – DISKETTE

The Event Driven Executive supports both the 4964 Diskette Storage Unit, and the 4966 Diskette Magazine Unit. Diskettes used with the Event Driven Executive can be Diskette 1 (single-sided), Diskette 2 (double-sided) or Diskette 2D (double-sided double-density) diskettes. EDX supports all diskettes formatted 256 bytes/sector and Diskettes 1 and 2 formatted 128 bytes/sector. Diskette 2D can only be used in the 4966 Diskette Magazine Unit.

### DEVICES SUPPORTED – DISK

The 4962 Disk Storage Unit and the 4963 Disk Subsystem are non-removable direct access storage devices, available in several capacities, with or without fixed-head capability. All models of both devices are supported by the Event Driven Executive.

For information on the physical layout of any of the disk/diskette storage devices, see the appropriate General Information manual for that device.

### DEVICES SUPPORTED – TAPE

The Event Driven Executive supports all models of the 4969 Magnetic Tape Subsystem utilizing 800 or 1600 bpi magnetic tape.

## Disk Volume Definition

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Disk I/O." IBM Series/1 Event Driven Executive System Guide (SC34-1702), "Direct Access Storage Devices" and "DISK Configuration Statement."

Event Driven Executive direct access storage has an hierarchical structure. The largest logical unit in the hierarchy is the volume. A volume is named contiguous area on disk/diskette, starting and ending on a cylinder boundary.

Disk devices are identified to the supervisor by the DISK system configuration statement at system generation time. The DISK statement defines the type of disk and hardware address of disks to be supported by the supervisor.

Volumes on disk are defined by the \$INITDSK utility. Before allocating any volumes, a user must initialize the disk device. This device initialization creates a volume directory which will contain control information about volumes that are subsequently allocated on the device. Once the volume directory is created, volumes can then be allocated. Before using a newly created volume, it must be initialized; that is, a directory must be created to contain the control information about data sets that will subsequently be allocated in the volume.

## Diskette Volume Definition

Logical volumes defined on disk devices exist on a non-removable storage medium. The names used to symbolically reference these disk volumes (EDX002, ASMLIB, etc.) are recorded in the volume directory of the device. Diskettes, being removable, also have the volume name written on them.

A DISK statement is used to define a diskette device (4964 or 4966), and to establish the device hardware address. This generates the physical device tables the supervisor requires to operate the device.

A logical volume on diskette encompasses the entire diskette. A diskette volume mounted on a 4964 device is considered to be a logical volume, as only a single diskette may be mounted and online at a time. With the 4966 Diskette Magazine Unit, up to twenty-three volumes may be online.

Diskette volumes are created (volume name written, directory created, etc.) by the \$INITDSK utility. As many volumes as required may be created.

## Tape Volume Definition

Each magnetic tape is a volume which is allocated by the \$TAPEUT1 utility. Only one volume is defined for each physical tape drive known to the system. The actual volume label is determined by the system when the tape is placed on line (\$VARYON).

## Data Sets

Data sets are members of a library. A data set is a named contiguous space whose length is determined by the user when the data set is created. Disk or diskette data sets are allocated by the user, using utility program \$DISKUT1, or, in some cases, automatically allocated by certain special purpose system utilities. Data sets may be defined with program organization or data organization, depending on what is to be stored. Program organization is used for data sets that will contain executable (loadable) Event Driven Executive programs. Data organization is used for work files (\$EDIT1N, \$FSEDIT, \$LINK, \$EDXASM, \$S1ASM work files), user source modules and application data sets.

Tape data sets are allocated by the user using the \$TAPEUT1 utility. Data sets can only be defined as data organization. Programs cannot be loaded and executed from tape.

## Records

When a data set is allocated, an entry is made in the directory of the logical volume in which the data set is defined. The directory entry will contain such information as the data set name, organization (program or data), location of the data set starting point within the volume, and the length of the data set, in records.

A record is 256 bytes in length, and is the smallest logical unit in the Event Driven Executive direct access storage hierarchy. A record is the basic unit that is accessed from user and system programs. Data sets are named, contiguous groups of 256-byte records.

## Record/Sector Relationship

Diskettes used as Event Driven Executive logical volumes can be formatted in 128-byte or 256-byte sectors. When formatted in 128-byte sectors, two diskette sectors constitute a single 256-byte logical record. On a 4962 disk, physical sectors and logical records are the same length, 256 bytes. The physical sector size on 4963 disks is 512 bytes, allowing two logical 256 byte records in each physical sector.

In all cases, user access to direct access storage is at the logical 256-byte record level. System routines compensate for physical sector/logical record mismatches, making the hardware differences between devices transparent to the user.

Figure 10-1 summarizes the direct access storage logical layout.

## PROGRAM STATEMENT DS= OPERAND

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "PROGRAM Statement."

Data sets accessed from user programs must be preallocated on disk or diskette (\$DISKUT1 utility), and must be named in the DS= keyword operand of the using program's PROGRAM statement. Figure 10-2 shows how the DS= operand is coded for data sets residing on the IPL or other logical volumes.

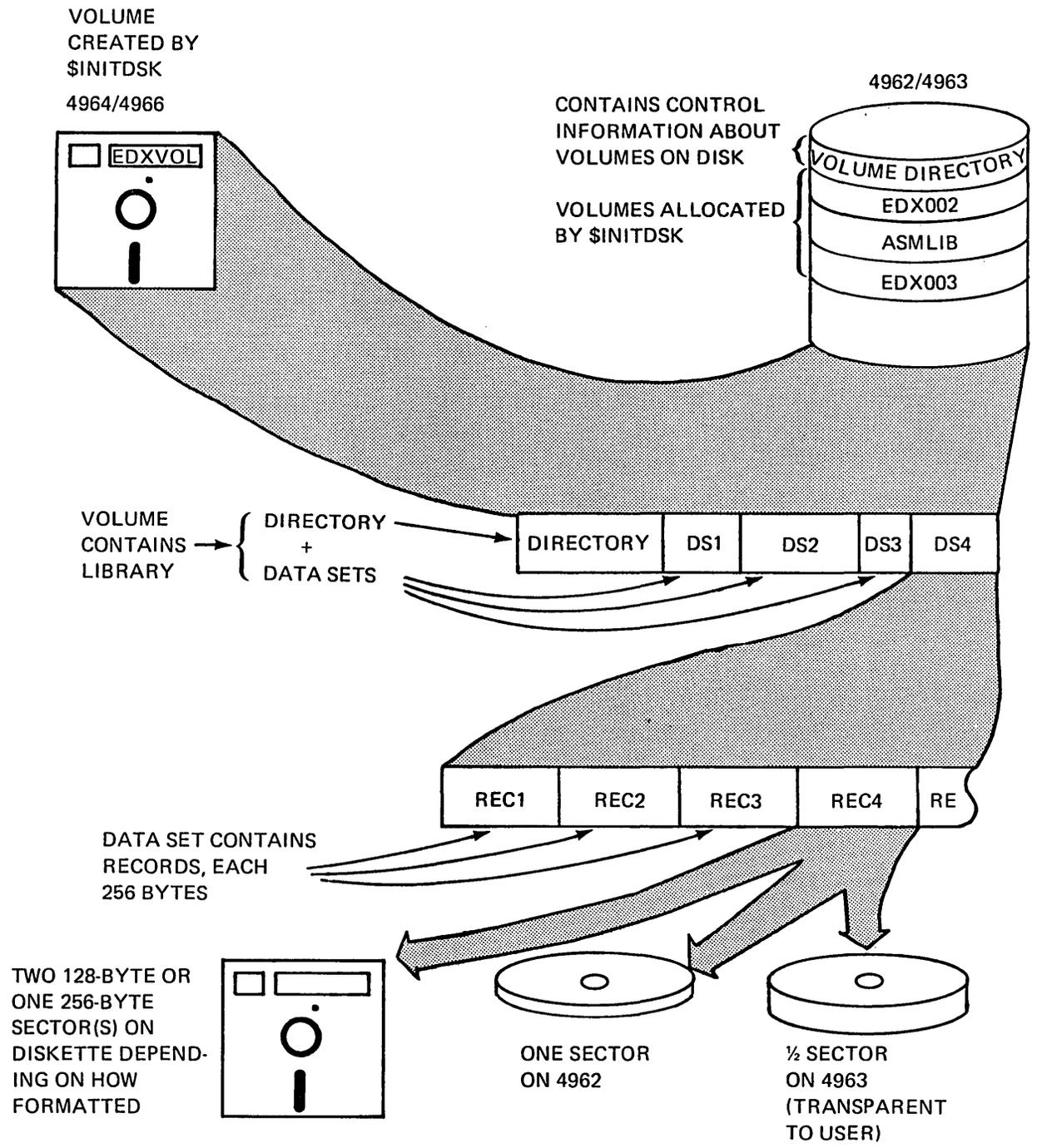


Figure 10-1. Disk/diskette logical layout

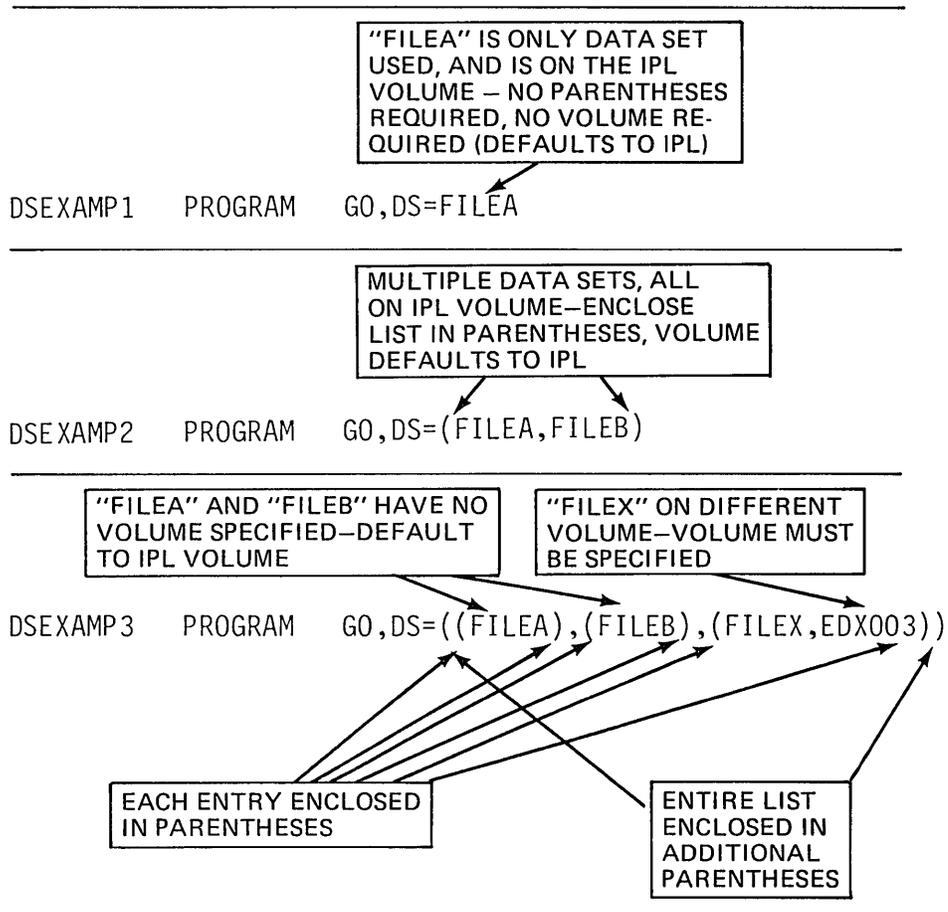


Figure 10-2. DS= operand

The IPL volume is the volume where the currently loaded (IPL) supervisor resides. The system will assume that data sets specified in the DS= operand list also reside on the IPL volume, unless a different volume is explicitly coded. Up to nine data sets may be coded in a DS= operand list.

At the time a program is loaded, the loader (\$LOADER) looks up all the data sets named in the PROGRAM statement's DS= operand list, and logically opens them for use by the program. If a named data set does not exist (was never allocated by \$DISKUT1), resides on a volume other than that specified in the DS= operand entry, or is program rather than data organization, the load operation is terminated and an error message results.

## READ/WRITE STATEMENTS – DISK/DISKETTE

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "READ", "WRITE."

The 256-byte records in data sets are transferred from disk/diskette to storage or storage to disk/diskette by READ and WRITE instructions. The format for READ and WRITE statements is illustrated in Figure 10-3.

DSx is the operand specifying which data set to use. The x in DSx is coded as an integer value between 1 and 9, and is a positional reference to one of the 9 possible data sets named in the DS= list of the PROGRAM statement.

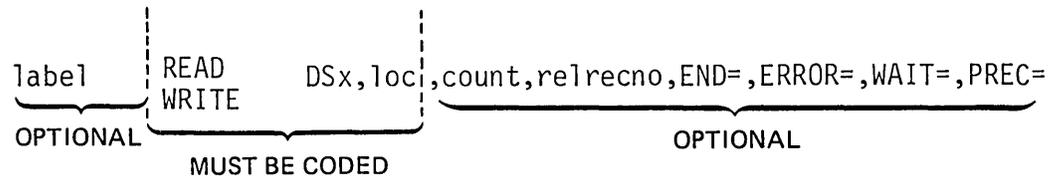


Figure 10-3. READ/WRITE format—disk/diskette

DS1 would refer to the first data set in the list, DS2 to the second, continuing through DS9, referencing the ninth data set defined.

The loc operand is coded as the label of the first byte of the (one or more contiguous) 256 byte storage area(s), into or from which the disk/diskette record(s) will be read/written.

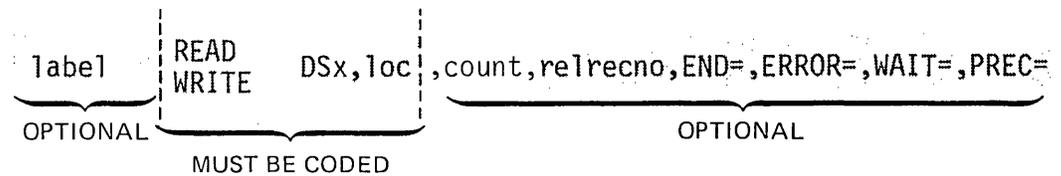


Figure 10-4. READ/WRITE count operand

The optional count operand is coded as an integer value (or as the label of a program location containing an integer value) indicating the number of 256-byte records to be read or written. The user must ensure that adequate storage is reserved (beginning at location loc) to accommodate the number of records specified in count. If count is not coded, the system will default the count operand to 1, indicating that a single record will be read or written. If count is set to 0, the READ or WRITE will not be performed (execute as a no-op), and execution will continue with the next sequential instruction following the READ/WRITE.

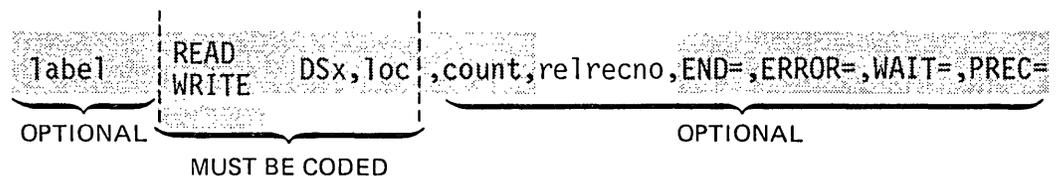


Figure 10-5. READ/WRITE relrecno operand

The relrecno operand is the relative record number (relative to the origin of the data set) to be read or written. It is coded as an integer, or as the label of a location containing an integer, which is the relative record number you want to access. The relrecno operand will default to 1 (indicating the first record in the data set) if it is left uncoded.

For each data set used by a program (DS1, DS2, etc.), the system maintains a next-record pointer. This pointer is an indicator of the next sequential record in the data set and, at the time a program is loaded (before disk/diskette I/O has been performed), has an initial value of 1. It is updated by +1 after each READ or WRITE in which;

- a. relrecno is not coded
- b. relrecno is coded as 0
- c. the location specified by the label in relrecno is equal to 0

Successive executions of READ/WRITE instructions in which relrecno has a value of 0 or is not coded will therefore result in sequential access of the data set; i.e., the relative record number of the next record read/written will automatically be 1 greater than the last record read/written. A READ or WRITE with relrecno coded as an integer greater than 0, or with the contents of the location specified by the label in relrecno greater than 0 does not disturb (increment) the next-record pointer.

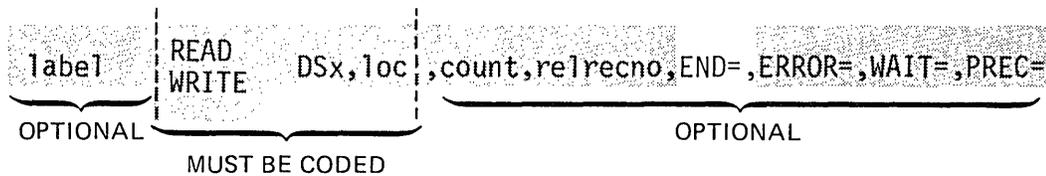


Figure 10-6. READ/WRITE END= operand

The END= keyword operand is coded with the label of the instruction that you wish control transferred to when an attempt to READ or WRITE a record outside the physical boundaries of the data set is detected. This condition may occur because of a normal end-of-data set condition (attempting to READ or WRITE the next sequential record in a data set, when the last record read or written was the last physical record in the data set), or may be caused by a program logic error (for example, a READ or WRITE with relrecno erroneously set to a negative value).

*Note:* A "logical-end-of-data" facility for READ operations is provided by a system subroutine called SETEOD. This subroutine will allow a user to set a given record number in a data set as the last logical record in that file. An attempt to read a record beyond the last logical record (although still within the physical boundaries of the data set) would result in a transfer to the label coded in the READ statement's END= keyword operand.

SETEOD is supplied as a system COPYCODE source module, and may be copied into user programs using the COPY assembler instruction.

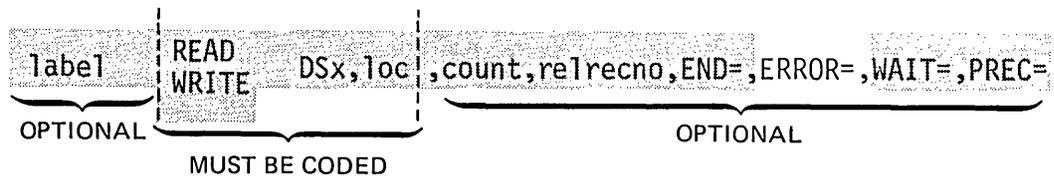


Figure 10-7. READ/WRITE ERROR= operand

The ERROR= keyword operand is coded with the label of the instruction that you wish to get control if an error is detected while executing a disk/diskette READ or WRITE operation. If END= is not coded and ERROR= is coded, an end-of-data set condition will result in a transfer to the ERROR= location. If END= is coded and ERROR= is not, all abnormal conditions other than end-of-data set will result in continuation of execution with the next sequential instruction following the READ or WRITE. If neither is coded, execution continues with the next sequential instruction in all cases.

After each disk/diskette READ or WRITE operation, a completion code is returned to the user program (see Reading Assignment for a description of completion codes). The completion code is placed in the task code word (taskname) of the task issuing the READ or WRITE, and is also placed in a system control block that may be referenced by the symbolic positional data set name (DS1, DS2, etc.). This completion code can be accessed and analyzed by the user program to determine if the operation was successful and, if not, why it failed.

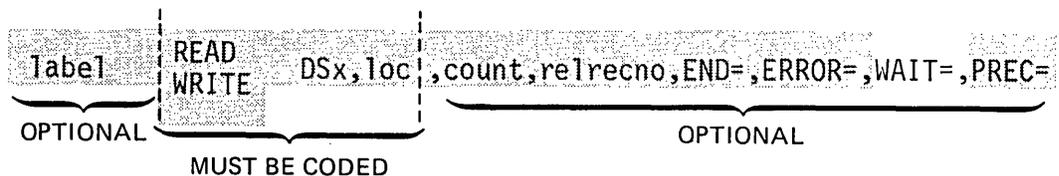


Figure 10-8. READ/WRITE WAIT= operand

While a disk/diskette I/O operation is executing, there is an implied wait for the issuing task. Task execution is suspended (the task is placed in a wait state) until the I/O is complete. If the WAIT= operand is coded as WAIT=NO, the wait does not occur; while the I/O operation is in progress, task execution proceeds with the next sequential instruction following the READ or WRITE, overlapping I/O with processing. Also, if WAIT=NO is coded, the END= and ERROR= keyword operands are not allowed. Checking for errors is entirely a user responsibility (completion code in taskname or DSx).

In addition, the user must issue an explicit WAIT instruction, waiting on the completion of I/O event. This is a predefined system event, and the associated ECB is referenced (in the operand of the WAIT statement) by the symbolic positional data set name (DS1, DS2, etc.) for the data set used. When the waited on ECB is posted complete, the I/O operation has finished, and the completion code is available for inspection.

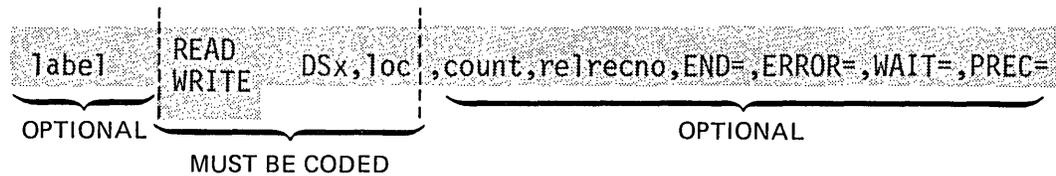


Figure 10-9. READ/WRITE PREC= operand

The PREC= keyword operand further defines the relrecno operand. If PREC= S (single precision) the relative record number is limited to a value of 32767. If PREC= D (double precision) is specified relative record numbers up to  $2^{31}-1$  can be specified. If no PREC= operand is specified the default assumed is S (single precision).

## READ/WRITE STATEMENTS – TAPE

Records in data sets are transferred from magnetic tape to storage or storage to magnetic tape by READ and WRITE instructions. As with disk/diskette, the DSx operand specifies the data set to be used. The format of the READ and WRITE statements is shown in Figure 10-10.

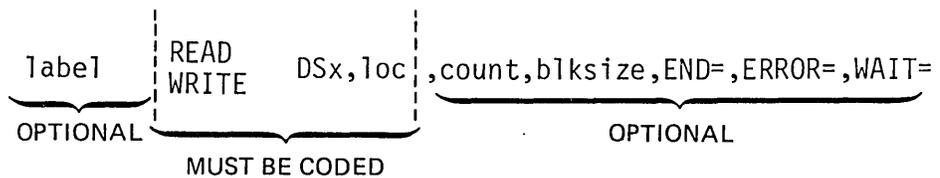


Figure 10-10. READ/WRITE format-tape

All the operands as discussed previously for disk/diskette apply to tape READ/WRITE operations. Tape records can be variable in length and are normally read sequentially; therefore, a blksize operand is used instead of the relrecno operand. BLKSIZE indicates the number of bytes to be read from or written to tape. If no BLKSIZE is specified, it will default to 256. Since the maximum size of a tape record is 32767 bytes, the PREC= operand does not apply.

## NOTE/POINT STATEMENTS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "NOTE", "POINT."

The system-maintained next record pointer changes value (increments) each time a READ or WRITE (without a user-specified relrecno greater than 0) is executed. Using the NOTE instruction, a user program can find out the current value of the next record pointer. The next record pointer may be set to a user-specified new value using the POINT instruction.

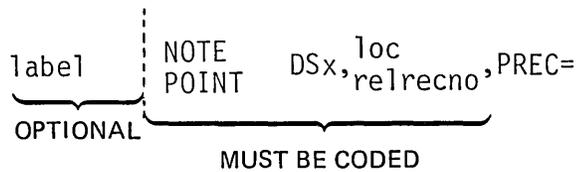


Figure 10-11. NOTE/POINT format

In Figure 10-11, the DSx operand is the symbolic positional reference to the data set whose associated next record pointer is to be retrieved (NOTE) or set (POINT). The second operand is coded as the label of a storage location that the NOTE instruction will move the current value of the next record pointer into, or that contains the new value which the POINT instruction will use to set the next record pointer. (When using the POINT instruction, the second operand may be coded as an integer value rather than the label of a storage location.)

## DISK/DISKETTE I/O CODING EXAMPLES

The programs depicted in the next four figures (Figure 10-12 through 10-15) are not meant to be practical examples of how to code disk/diskette I/O operations in a user program. They are intended only to illustrate some of the concepts already discussed.

In Figure 10-12, the READ instruction at location GO will execute as a no-operation. Execution will continue with the instruction following the READ, and no I/O is performed. The count operand is coded as storage location CTR. When the program is first loaded, location CTR contains zero, and a zero count indicates no records are to be read (or written, for a WRITE instruction).

```

DISKPGM PROGRAM GO,DS=WORKFILE
GO      READ   DS1,BUFF,CTR,END=ENDOUT,ERROR=E1
      .
      .
R1      READ   DS1,BUFF,END=ENDOUT,ERROR=E1
      .
      .
SET7    POINT  DS1,7
      MOVE   CTR,3
R2      READ   DS1,BUFF,CTR,END=ENDOUT,ERROR=E1
      PROGSTOP
ENDOUT  END-OF-DATA SET
      ROUTINE
E1      ERROR ROUTINE
BUFF    BUFFER 768,BYTES
CTR     DATA  F'0'
      ENDPROG
      END

```

Figure 10-12. Count operand use

The READ at location R1 has no count operand coded, so count defaults to 1, indicating a single record will be read. Since relrecno is not coded, the relative record number defaults to the current value of the next record pointer. The next record pointer has not yet been altered, and is therefore at its initial value of 1, indicating the first relative record in the data set. The READ at R1 will read the first record in WORKFILE into the first 256 bytes of the 768 byte area BUFF. After the I/O operation, the next record pointer is incremented to 2 (automatic system function).

The POINT instruction at location SET7 changes the next record pointer to point to the seventh relative record in the data set. The MOVE which follows sets location CTR to a value of 3. When the READ at R2 is executed, three 256 byte records (count = CTR = 3), beginning with relative record number 7 (relrecno defaults to next record pointer which was set to 7) will be read into storage, beginning at location BUFF. After the operation, the next record pointer will have a value of 10.

In Figure 10-13, all count operands are left uncoded, so all READ operations will be single record reads (default count = 1). In the first READ (location GO), relrecno is coded as location RECNBR, which has an initial value of 2. The second relative record in WORKFILE will be read into BUFF. The ADD instruction following the READ updates the user-maintained relative record number in RECNBR by adding 3. When the READ at R2 is executed, relative record number 5 will be read into BUFF.

The MOVE operation preceding the READ at R3 sets the relrecno location RECNBR to zero. A zero relrecno value causes a default to the next record pointer maintained by the system.

```

DISKPGM  PROGRAM  GO,DS=WORKFILE
GO       READ     DS1,BUFF,,RECNR,ERROR=ERRRUTN,END=OUT
        ADD     RECNR,3
        .
        .
R2       READ     DS1,BUFF,,RECNR,ERROR=ERRRUTN,END=OUT
        .
        .
        MOVE    RECNR,0
R3       READ     DS1,BUFF,,RECNR,ERROR=ERRRUTN,END=OUT
        .
        .
R4       READ     DS1,BUFF,ERROR=ERRRUTN,END=OUT
        .
        .
P1       PROGSTOP
OUT      [END-OF-DATA SET]
        [ROUTINE]

ERRRUTN [ERROR ROUTINE]

BUFF     BUFFER   256,BYTES
RECNR    DATA    F'2'
        ENDPROG
        END

```

Figure 10-13. "relrecno" operand use

The two previous READ operations (at GO and R2) both used a user-defined relrecno value greater than zero, so the next record pointer was not affected, and is still at its initial value of 1. The READ at R3 will therefore read the first relative record in WORKFILE, because the MOVE operation preceding sets RECNR to zero.

The READ at R4 has no relrecno coded, and will also default to the next record pointer for a relative record number. This READ will read relative record number 2, since the next record pointer was incremented by 1 after the preceding READ at R3.

In Figure 10-14, all count and relrecno operands are left uncoded, so all READ commands will read a single record, and the next record pointer will be used for the relative record number.

The READ statement at GO has both END= and ERROR= operands coded. An end-of-data set condition will cause a transfer to location ENDR, and an error condition will result in execution of the instructions beginning at ERTN. If the operation is successful, relative record number 1 will be read into BUFF.

In the READ statement at R2, only the END= operand is used. Error checking is therefore a user responsibility, and is performed in this example by the IF statement immediately following the READ. The symbolic positional data set name, DS1, is checked for a completion code of -1. A -1 indicates a successful or normal operation. If the completion code is other than -1, control is transferred to the error routine at ERTN. If the operation was successful, relative record number 2 would be read.

```

DISKPGM  PROGRAM  GO,DS=WORKFILE
GO        READ    DS1,BUFF,END=ENDR,ERROR=ERTN
.
.
.
R2        READ    DS1,BUFF,END=ENDR
          IF      (DS1,NE,-1),GOTO,ERTN
.
.
.
R3        READ    DS1,BUFF,ERROR=E0
.
.
.
R4        READ    DS1,BUFF
          IF      (DS1,NE,-1),GOTO,E0
.
.
.
DONE      PROGSTOP
ENDR      PRINT OUT "END
          OF DATA SET" MSG
.
.
.
          GOTO    DONE
E0        IF      (DS1,EQ,10),GOTO,ENDR
ERTN      PRINT OUT "DISK
          ERROR" MSG
.
.
.
          GOTO    DONE
BUFF      BUFFER  256,BYTES
          ENDPROG
          END

```

Figure 10-14. END= and ERROR= use

The ERROR= operand is coded in the READ statement at R3, but the END= is not. An end-of-data set condition will therefore be considered an error, and will cause a transfer to the label coded in the ERROR= operand, location E0. When END= is not coded, but you do not wish to treat end-of-data set as an error, the specific condition code that indicates end-of-data set must be checked for in the error routine. The IF statement at location E0 checks for a completion code of 10, which is the completion code signifying an end-of-data set (relative record number outside range of data set) condition. If the code is 10, control transfers to the end-of-data set routine at ENDR, rather than continuing execution of ERTN. Relative record number 3 is read if normal operation occurs.

The READ at R4 has neither END= nor ERROR= coded. Operation is the same as the previous READ at R3, except that the user must check for abnormal completion; there is no automatic transfer to an error routine, as is provided by the ERROR= operand. The completion code is checked by the IF statement following the READ, and transfers to E0 (as did the ERROR=E0 in the READ at R3) if other than normal completion is detected. Normal completion results in a read of relative record number 4.

Figure 10-15 illustrates the use of the WAIT= operand. The READ at location START is the same as the READ statements you are already familiar with. It will read a single record (count defaults to 1), the first relative record in data set WORKFILE (relrecno defaults to next record pointer = initial value of 1), into BUF1. If an error occurs, the ERROR= operand will transfer control to E1, the start of the error routine. (END= is not required because, by definition, if WORKFILE exists, it has at least one record in it. Since this is a read of the first record in WORKFILE, end-of-data set will not occur.)

While the READ at START is in progress, task DISKPGM is in a wait state (WAIT= operand not coded — default is WAIT=YES). After successful completion of the READ, the MOVE at location SETUP is executed, moving the 256 byte record in BUF1 into WRKAREA (128 words = 256 bytes).

Now a second READ is issued (location R2), with the WAIT= operand coded as WAIT=NO. Since the READ at START used the next record pointer for a relative record number, it now has a value of 2. The READ at R2 will therefore read relative record number 2 into BUF1, updating the next record pointer to 3 upon successful completion.

While the READ operation at R2 is in progress, execution of task DISKPGM continues, because the WAIT=NO operand prevents the implied wait for I/O completion from taking effect. While the next sequential record (relative record 2) is being read into BUF1, the program is operating on the data in the previous record, which is now in WRKAREA. Program execution is overlapping with the I/O.

```

DISKPGM  PROGRAM  START,DS=WORKFILE
BUF1     BUFFER   256,BYTES
WRKAREA  DATA    128F'0'
START    READ     DS1,BUFF1,ERROR=E1
SETUP    MOVE     WRKAREA,BUFF1,128
R2       READ     DS1,BUFF1,WAIT=NO
.
.
.
PROCESS THE DATA IN
"WORK AREA"
.
.
W1       WAIT     DS1
IF1      IF       (DS1,EQ,-1),GOTO,SETUP
IF2      IF       (DS1,EQ,10),GOTO,OUT
.
.
.
E1       PRINT DISK ERROR
          MESSAGE
.
.
.
STOP     PROGSTOP
OUT      PRINT END OF DATA
          SET MESSAGE
.
.
.
GOTO     STOP
ENDPROG
END

```

Figure 10-15. WAIT=NO

When WAIT=NO is coded, as illustrated in the READ at R2, the ERROR= and END= operands cannot be used. Error checking is therefore entirely a user responsibility. The I/O operation completion code is not available until the I/O operation is finished. To find out when the I/O is complete and the completion code is available, and also to resynchronize processing with I/O, the user must issue a WAIT on the completion of I/O event.

The WAIT at location W1 uses the symbolic positional data set name DS1 as the event name. The ECB is not coded, because it already exists in the TCB established by the PROGRAM statement. When the READ operation at R2 completes, the completion code is posted in location DS1. DS1 is the symbolic address of the first word of the associated ECB, and therefore the completion of I/O event is marked as having occurred.

After the WAIT, execution continues with the IF statement at location IF1. If the I/O completed normally (condition code = -1), control is transferred to SETUP, which moves the new record into the work area. The READ at R2 starts the read of the next sequential record into BUF1, and the entire process continues to repeat until all records have been processed (end-of-data set) or an error occurs.

If other than a normal completion is detected at IF1, the IF at IF2 executes. An end-of-data set condition (completion code = 10) will cause a transfer to location OUT, the end-of-data set routine. Any other completion code is an error, and execution will continue with the error routine E1, immediately following the IF.

## TAPE I/O EXAMPLES

For comprehensive sample programs of magnetic tape operations, review the examples in the "Tape Management" section of the System Guide (SC34-1702).

## LOAD-TIME DATA SET DEFINITION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "LOAD", "PROGRAM."

In all of the disk/diskette I/O examples thus far, data sets to be used by a program are named in the DS= list of the PROGRAM statement. This is adequate for very stable applications, where the program always uses the same data sets, and the names of those data sets are known at the time the program is written.

A stable situation is not always possible. At the time a particular program is being coded, data set naming conventions may not yet have been established, and data set names therefore would not be known. Also, the program could be a generalized file routine, designed to perform certain updating or maintenance functions on any of several similar data sets, a different data set (and data set name) each time the program is executed.

By coding ?? in place of a data set name (in the DS= list of the PROGRAM statement), data set names can be specified at the time a program is loaded for execution, rather than when it is coded. In Figure 10-16, the first entry in the DS= list is coded as ??, and the second entry as the data set name FILEA.





Data set names that *are* in the DS= list of the loading program's PROGRAM statement can be passed using the actual name, or by using the symbolic positional reference DSx. At LD2 in PROGA (Figure 10-17), PROGB is again loaded, passing the data set DS1. This refers to the first entry in the DS= list in PROGA's PROGRAM statement, which is coded as ??. Again assuming this data set name was supplied by a terminal operator when PROGA was loaded, that same name will be passed through to PROGB, becoming the data set used by PROGB for the WRITE operation. If DS2 instead of DS1 were coded, FILEA would have been passed.

When programs using disk/diskette I/O are loaded as overlays, *all* names of data sets used by the overlay program must be passed by the loading program, and the data set names that are passed *must* be entries in the DS= list of the loading program's PROGRAM statement. In Figure 10-18, the PROGRAM statement for PROGA defines PROGB as an overlay program (PGMS=PROGB). The LOAD statement at LD3 will load PROGB as an overlay, because the program name specified is PGM1, a positional reference to the PGMS= list. PROGB uses two data sets, so two data set names are passed to PROGB in the LOAD statement's DS= list: DS2 and DS1, which reference FILEA and ?? in the DS= list for PROGA. When passing data set names to an overlay program, the LOAD statement must use the DSx positional references.

All data sets used by an overlay program must be passed to the overlay by the loading program, and therefore all data set names in the DS= list of the PROGRAM statement of a program loaded as an overlay are treated as though they were ?? entries. For example, if PROGB is loaded as an overlay, FILEB will not be used, unless it is passed by the LOAD statement in the loading program.

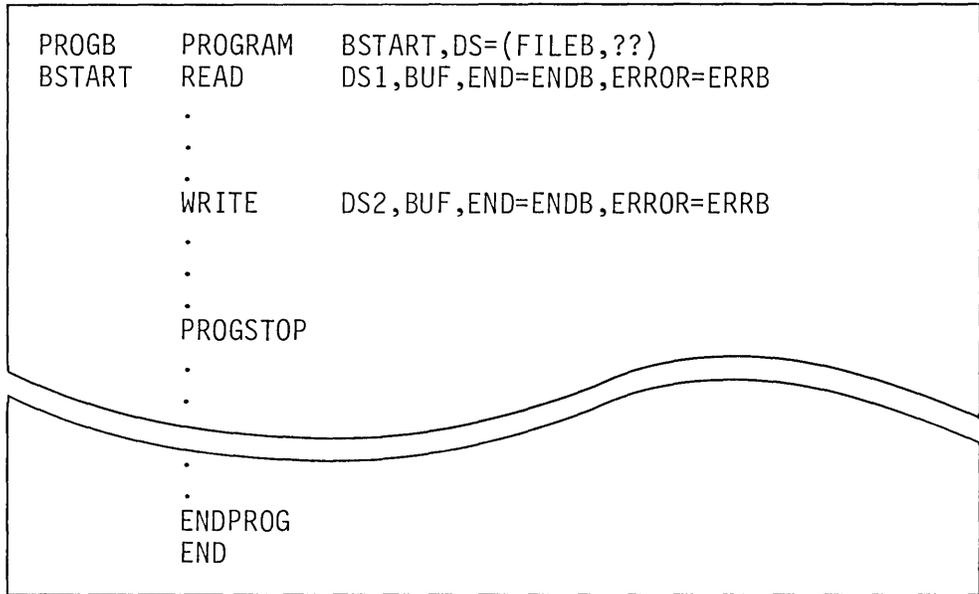
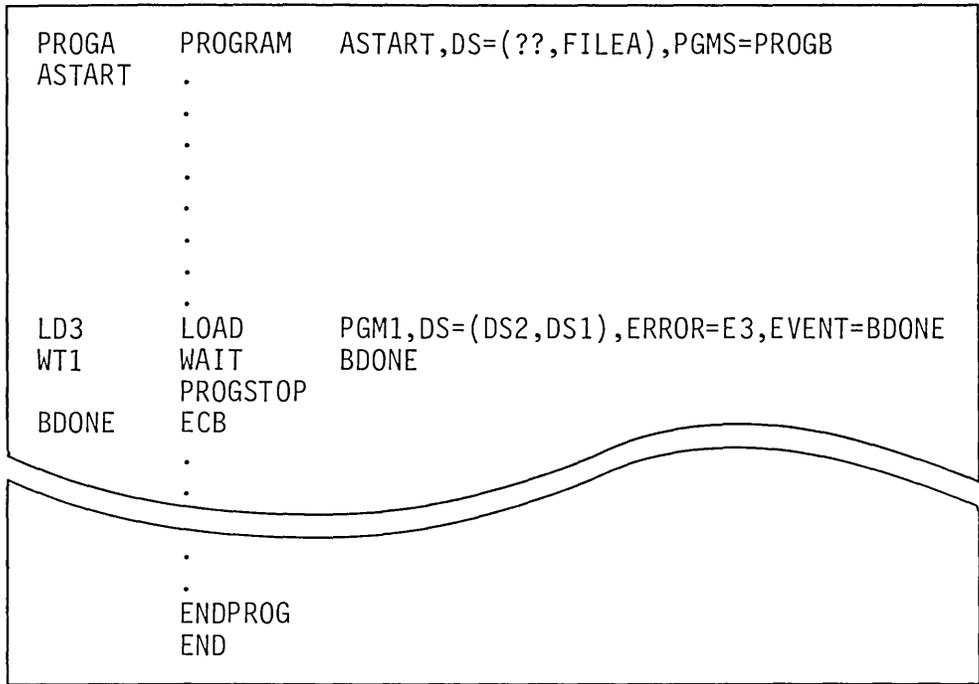


Figure 10-18. Overlay load -- data set passing

In Figure 10-18, if the terminal operator loading PROGA (\$L PROGA) responds to the DS1=(NAME,VOLUME): prompt by entering FILEC, PROGA will execute as though the DS= list in the PROGRAM statement were coded DS=(FILEC,FILEA). In the DS= list of the LOAD at LD3, the first entry is DS2. This first position in the LOAD statement's DS= list corresponds to the first position in the DS= list for PROGB. The DS2 references the second entry in the DS= list of PROGA's PROGRAM statement, which is coded as FILEA. The data set name FILEA is therefore passed to PROGB as the first entry of the DS= list in the PROGRAM statement for PROGB. Similarly, the second entry in the LOAD statement's DS= list will pass FILEC, the DS1 data set name entered by the operator, to the second entry in the DS= list for PROGB. PROGB will execute as though the DS= list in the PROGRAM statement were coded as "DS=(FILEA,FILEC)". The READ will be from FILEA, and the WRITE to FILEC.

## DISK/DISKETTE I/O REVIEW EXERCISE—QUESTIONS

1. How many volumes may be defined on a 4962/4963 Disk Storage Unit? \_\_\_\_\_
2. Which of the following choices, when used to complete the statement below, makes the statement *not* true?  
“The DS= list in a PROGRAM statement . . .
  - a. . . . must contain an entry for each data set used by the program.”
  - b. . . . may contain up to nine entries.”
  - c. . . . may specify data sets resident on other than the IPL volume.”
  - d. . . . is used to define the names of any overlay programs that may be loaded by the program.”
  - e. . . . may have entries for data sets that will not be defined until load time.”

All of the remaining “Questions for Review” refer to the coding example in Figure 10-19.

```

PROG1  PROGRAM  GO,DS=(DSET1,DSET2,DSET4,DSET9),PGMS=P2
GO     READ     DS3,BUFA,NBR,RCRD,END=E1,ERROR=E2
RD2    READ
IF1    IF       (____,__,_),GOTO,E1
IF2    IF       (____,__,_),GOTO,E2
N1     NOTE     DS3,DS3VAL
LD1    LOAD     P2,DS=(____,____),ERROR=LDERR
LD2    LOAD     _____,DS=(____,____,____),ERROR=LDERR
      PROGSTOP
BUFA   BUFFER   _____,BYTES
DS3VAL DATA   F'0'
NBR    DATA   F'2'
RCRD   DATA   F'5'
      .
      .
      .
      ENDPROG
      END

```

```

P2     PROGRAM  PGO,DS=(??,DSET3,??)
PGO    READ     DS3,BUFF
      .
      .
      .
PR2    READ     DS1,BUFF
      .
      .
      .
PR3    READ     DS2,BUFF
      .
      .
      .
      PROGSTOP
BUFF   BUFFER   128
      ENDPROG
      END

```

Figure 10-19. Review problem

3.
  - a. How many records will be read by the READ at location GO?
  - b. What is the name of the data set used?
  - c. What is the relative record number of the first record that will be read?
  - d. What should be coded as the first operand of the BUFFER statement at location BUFA?

Answers:

- a. \_\_\_\_\_
- b. \_\_\_\_\_
- c. \_\_\_\_\_
- d. \_\_\_\_\_

4. Code the READ at RD2 to read a single record (let count take default) into BUFA. The record should be the first relative record (let relreco take default) in data set DSET4. Do not code the END= or ERROR= operands. Code the IF at IF1 to check for end-of-data set condition, and the IF at IF2 to check for other errors.

Answer:

```

.
.
RD2  READ
IF1  IF      (____,____,____),GOTO,E1
IF2  IF      (____,____,____),GOTO,E2
.
.

```

5. After executing the NOTE instruction at N1, what will be the value of location DS3VAL?

Answer: \_\_\_\_\_

6. Code the LOAD instruction at location LD1 so that when program P2 executes, the READ at PGO will use data set DSET5, the READ at PR2 will use DSET9, and the READ at PR3 will read from DSET3.

Answer:

```

.
.
LD1  LOAD   P2,DS=(____,____),ERROR=LDERR
.
.

```

This page intentionally left blank.

7. Code the LOAD at location LD2 to load P2 as an overlay program. In program P2, the READ at PGO should use DSET1, the READ at PR2 data set DSET2, and the READ at PR3, data set DSET4.

Answer:

```
.  
. LD2  LOAD  _____,DS=(_____,_____,_____),ERROR=LDERR
```

8. The LOAD at LD2 is a load of an overlay program. What must be added to PROG1 to ensure the proper termination-of-execution sequence between P2, the overlay program, and PROG1, the loading program?

Answer: \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

## DISK/DISKETTE I/O REVIEW EXERCISE—ANSWERS

1. Each 4962/4963 may have as many volumes as required defined, within the physical size limitations of the device.
2. All choices except choice "d" will complete the statement truthfully. The "PGMS=" keyword operand is used to define the overlay programs.
3.
  - a. 2 records will be read (count=NBR=2)
  - b. DSET4 will be used. DSET4 is the third entry in the DS= list, and is referenced by DS3 in the READ at GO.
  - c. relative record number 5 (relrecno=RCRD=5)
  - d. 512 or more, because two 256 byte records are being read (NBR=2).
4.

```
RD2  READ  DS3,BUFA
IF1  IF     (DS3,EQ,10),GOTO,E1
IF2  IF     (DS3,NE,-1),GOTO,E2
```
5. DS3VAL will contain 2, because the next record pointer is updated by +1 following the READ at R2.
6. LD1 LOAD P2,DS=(DS4,DSET5),ERROR=LDERR
7. LD2 LOAD PGM1,DS=(DS2,DS3,DS1),ERROR=LDERR
8. The LOAD at LD2 should have the EVENT= operand coded, declaring an event name. An ECB with that event name should also be coded, and a WAIT on that event name should occur prior to the PROGSTOP.

## Section 11. Terminal I/O

**OBJECTIVES:** After completing this section, the student should be able to:

1. Describe roll screen and static screen operation
2. Use PRINTTEXT, PRINTIME, PRINDATE, and PRINTNUM instructions to display data on a terminal
3. Use READTEXT and GETVALUE instructions to read data from a terminal
4. Understand the purpose of specialized terminal instructions such as QUESTION, TERMCTRL, etc.

The Event Driven Executive terminal support is designed to be as device independent as possible. With few exceptions, the user need not be concerned with what type of device is being driven by terminal functions coded in the program. The same sequence of terminal output instructions, for instance, may be used to print data on a matrix or line printer, on a locally attached TTY device or a remote ACCA terminal, or to display the data on an electronic display screen device.

The specific terminal support applies to the IBM 4978, 4979 and 3101 displays. The 3101 Models 10, 11, 12 and 13 operate in character (roll screen) mode. A 3101 operating in character mode, attached via the teletypewriter adapter card, can be used as the system console. The 3101 Models 20, 21, 22 and 23 can operate in either character mode or block (static screen) mode. To be used as a static screen (block mode) device, the 3101 M2 must be attached via the single line or multiline asynchronous communications adapter card. Discussions in this section which refer to a 3101 operating in block mode will be designated as 3101 M2.

### TERMINAL STATEMENT

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive System Guide (SC34-1702), "TERMINAL Configuration Statement."

Terminals are defined to the system using the TERMINAL system configuration statement. This statement generates system control blocks and tables containing the logical and physical variables necessary to operate the terminal. Among the physical variables described in the TERMINAL statement operands are the type of terminal (TTY, printer, display, etc.), its hardware address, the type of transmission code used, and other hardware related parameters unique to the device being defined.

The high degree of device independence is achieved in part by treating all terminals as though they were line printers, differing only in their page sizes (forms length) and margin settings, also defined by TERMINAL statement operands.

## Roll Screens

The page size for an IBM 4978/4979/3101 terminal is 24, the maximum number of lines that can be displayed on the screen. The 4978/4979/3101 Displays can be operated as roll screen or static screen devices (SCREEN= operand in TERMINAL statement). A roll screen device operates in much the same way as a typewriter. Assuming a blank screen (clean page in typewriter) to start, data is displayed line by line, beginning with line 0 at the top of the screen and continuing through line 23 at the bottom of the screen, just as a typewritten page is filled from top to bottom. When a page being typed is full, the completed page is removed, a clean page is inserted, and typing continues at the top of the new page. When a roll screen device's screen is full (all 24 lines used), an attempt to display the next line results in removal of the old screen (screen is erased) and display of the new line on line 0, at the top of the screen.

## NHIST= Operand

Unlike a typewriter, the display is not a hardcopy device, and therefore the information on the old screen (previous page) cannot be referred to after it has been erased. If an operator entry is expected and the operator prompts describing that entry were displayed on a now-erased previous screen, time could be wasted in looking up the input request in a reference book, or in requesting that the program repeat the display of the prompt.

This potential problem is avoided by coding the NHIST= operand of the TERMINAL statement to reserve part of the screen as a history area. NHIST= is the number of history lines you wish to reserve. For example, if NHIST=12 is coded, the top twelve lines of the screen are reserved for a history area (physical lines 0 through 11), and the bottom twelve lines (physical lines 12 through 23) as a work area, operating in the normal roll screen fashion. (The 4979 Display supported by the starter system is defined with NHIST=12, and NHIST=12 will be the default for user defined 4978/4979 displays if NHIST= is left uncoded. NHIST defaults to 0 for 3101.)

Since all terminals, including electronic display screens, are treated logically as printers, forms control commands are used to position displayed output on a screen, just as lines and spaces may be skipped on a printout to position a print line on a page. Although physically (with NHIST=12) the work area occupies lines 12 through 23, logically, for purposes of forms control interpretation, they are treated as lines 0 through eleven. Display information directed to line 0 will be displayed on physical line 12, the top of the work area.

Again beginning with a blank screen, successive lines are displayed starting at the top of the work area, and continuing to the bottom of the screen. With the work area full, an attempt to display the next line will cause:

1. the information displayed in the "work area" to be moved up into the "history area", (physical lines 0 through 11).
2. the "work area" (lines 12-23) to be erased
3. display of the new line on physical line 12, the top of the work area.

Each time the work area is exceeded, the information displayed there is moved up into the history area, thereby retaining some past history for viewing. The work area and history area do not have to be of equal size; you may code NHIST= to retain as few as 0 lines of previous data, or as many as 23 lines.

## Static Screens

Terminals operated as roll screen devices are usually used in an interactive mode, to communicate between a program and an operator. Operator prompts and their associated responses are exchanged on a line by line basis. The display of a new line, or the read of an operator entry is usually initiated by the operator pressing a terminal control key such as ENTER or one of the program function keys, indicating that the operation can proceed. A common example is the series of prompts and replies that are exchanged between program and operator when using the Event Driven Executive utilities.

When a 4978/4979/3101 M2 Display is defined as a static screen device (SCREEN= operand in TERMINAL statement), the screen is treated as a page of information. The screen may be formatted with pre-determined operator prompts (input field names), and these areas may be designated as "protected", preventing accidental overlay by input data. The input fields of a static screen are usually filled in by the operator without interaction with the program. Terminal operation keys such as TAB, BACKSPACE, or the cursor positioning keys are used to move the cursor to the required input field positions.

When all required input fields have been entered, the operator presses the ENTER key (or a designated Program Function key) to signal the program that the page is complete. The program then reads all the information on the screen, erases the screen, and displays a new page (screen with prompts, but blank input fields) for the operator to fill.

Terminals operated as static screen devices must be either IBM 4978, 4979 or 3101 M2 Displays, as some of the specialized instructions used with static screens can be interpreted only by the 4978/4979/3101 M2 hardware. Other electronic display screen devices and, of course, all hardcopy terminals, are operated as roll screens.

## ENQT/DEQT INSTRUCTIONS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "ENQT", "DEQT."

When a program is loaded from a terminal, that terminal is dynamically designated by the system as the terminal to be used by terminal I/O instructions in the program. Each terminal I/O instruction automatically has exclusive use of the terminal during the execution of that individual operation; only one task at a time is allowed to perform I/O on the terminal.

If more than one task is using the terminal, terminal operations from different tasks could become interspersed. In cases where this is undesirable, the ENQT (enqueue terminal) facility may be used to reserve the terminal for the exclusive use of a task, thereby preventing other tasks from using the terminal until the task issuing the ENQT releases it (DEQT).

label      ENQT      name, BUSY=  
OPTIONAL    MUST BE CODED    OPTIONAL

Figure 11-1. ENQT format

If ENQT is coded without the optional name operand, the default is to the terminal that loaded the program. The task issuing the ENQT will acquire exclusive control of the loading terminal, and will retain control until executing a DEQT instruction. If the terminal is busy (enqueued by another task) when the ENQT is executed, the task issuing the ENQT is placed in a wait state, queued up waiting for the terminal to become available. If you do not wish to be queued if the terminal is busy, the BUSY= operand should be coded with the label of the instruction to which you wish control transferred.

The ENQT may also be used to gain exclusive control of a terminal other than the loading terminal. The symbolic name assigned to a terminal is the name coded as the label of the TERMINAL statement defining the device. Coding a name in the label field automatically defines the terminal to the system as a global resource that may be enqueued by user programs (ENQT). There are three symbolic terminal names that have special significance, as they are used by the supervisor or system utility programs:

1. **\$\$SYSLOG** this is the name of the system logging device or operator station, and must be defined in every system. In the system configuration statements used to generate the supplied supervisor, \$\$SYSLOG is the label of a TERMINAL statement defining a 4978 Display.

2. **\$\$SYSLOGA** This is the name of the alternate system logging device. In the event that unrecoverable errors prevent use of **\$\$SYSLOG**, the system will use the **\$\$SYSLOGA** terminal as the system logging device/operator station. If defined (**\$\$SYSLOGA** is optional), this device should be a terminal with keyboard capability, not just a printer. The supplied supervisor **\$\$SYSLOGA** terminal is a TTY device.
3. **\$\$SYSPRTR** This is the name of the system printer, and is also optional. If defined, the output from some system programs will be directed to this device. In the supplied supervisor, **\$\$SYSPRTR** is defined as a 4974 matrix printer.

In addition to being used by the system, these devices may also be enqueued (ENQT) by user programs. In Figure 11-2, the ENQT/DEQT coding example refers to the terminals defined in the **TERMINAL** configuration statements shown at the top of the illustration. For simplicity, only the required **TERMINAL** statement operands are coded; all other operands are default values.

```

.
.
$$SYSLOG  TERMINAL  DEVICE=4979,ADDRESS=04
$$SYSPRTR TERMINAL  DEVICE=4974,ADDRESS=01
$$SYSLOGA TERMINAL  DEVICE=TTY,ADDRESS=00
DSPLY1    TERMINAL  DEVICE=TTY,ADDRESS=10,END=YES
.
.
.
.

```

TERMTASK	PROGRAM	START
START	ENQT	
	.	
	.	
D1	DEQT	
E2	ENQT	\$\$SYSPRTR,BUSY=E3
E3	ENQT	\$\$SYSLOG
	.	
	.	
D2	DEQT	
	PROGSTOP	
	ENDPROG	
	END	

Figure 11-2. ENQT/DEQT operation

Assuming that the loading terminal is the TTY device **DSPLY1**, the **ENQT** instruction at location **START** will acquire exclusive control and retain control until execution of the **DEQT** at **D1**. No name operand is coded for the **ENQT**, so the loading terminal **DSPLY1** is enqueued, thereby preventing other tasks from using **DSPLY1**.

The ENQT at E2 is directed at the 4974 matrix printer, \$SYSPRTR. If the matrix printer is already in use (enqueued), control is transferred to the next instruction at location E3 (BUSY=E3). This is an attempt to enqueue the 4979 display terminal \$SYSLOG. If \$SYSLOG is already enqueued, TERMTASK will be placed in a wait state, waiting until the terminal becomes available. In effect, the two ENQT statements at E2 and E3 may be interpreted as "try to get the system printer; if it is in use, get \$SYSLOG instead and use it."

If the ENQT at E2 executes successfully, acquiring control of \$SYSPRTR, the ENQT at E3 will execute as a no-op. When an ENQT for a given terminal has successfully executed and enqueued that terminal, ensuing ENQTs issued by the same task directed to terminals other than the terminal already enqueued are ignored. The system allows any one task to enqueue only a single terminal at a time. To switch from an already enqueued terminal to a different terminal, a DEQT must be issued before the ENQT for the new device is executed. DEQT commands are non-specific (no "name" operand), acting upon whatever terminal is currently enqueued by the issuing task.

## IOCB STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "IOCB."

One of the system control blocks generated by assembly of the TERMINAL system configuration statement is called an Input Output Control Block (IOCB). A terminal IOCB contains information such as the terminal's forms configuration (page size, margins), operating mode (static, roll), and history area size (NHIST= operand). A terminal is not restricted to the values coded for these parameters in the TERMINAL statement; they can be dynamically changed by a user program.

In Figure 11-3, a 4979 Display called DSPLY1 is defined in the TERMINAL statement at the top of the illustration. As you know from the previous discussion of roll screen operation, the NHIST= default value (for 4978/4979 Displays) is 12, dividing the screen into a history area and a work area of twelve lines each.

In TERMPROG (Figure 11-3), assume the user wants a screen that operates so that each new line is displayed on the last (bottom) line of the screen, forcing the previously displayed 24 lines up one for each new line displayed. This will cause the screen to act as a continuous scroll, with each new line forcing the oldest previous line off the screen at the top.

```
.  
. DSDPLY1    TERMINAL DEVICE=4979,ADDRESS=20  
.  
.
```

TERMPROG	PROGRAM	SCROLL
NEWHIST	IOCB	DSDPLY1,NHIST=23
SCROLL	ENQT	NEWHIST
	.	
	.	
DONE	DEQT	
	PROGSTOP	
	ENDPROG	
	END	

Figure 11-3. IOCB/ENQT

To operate in this way, a history area of 23 lines is required, leaving a one line work area for new entries. At location NEWHIST is a user-coded IOCB, which references terminal DSDPLY1, and defines NHIST= as 23. The ENQT at SCROLL references the IOCB label NEWHIST. Execution of the ENQT acquires exclusive control of, and puts the user-coded IOCB in effect for, the named terminal, DSDPLY1. (If no terminal name is coded, the system will default to the loading terminal.) Until execution of the DEQT at DONE, DSDPLY1 will operate with NHIST=23. The DEQT will cause DSDPLY1 to revert back to the IOCB values generated by the TERMINAL system configuration statement.

In the same manner, 4978/4979 Displays that are defined in TERMINAL statements as roll screen devices (SCREEN= default is ROLL) may be dynamically enqueued for static screen operation by a user program. Because Event Driven Executive system and utility programs expect a roll screen configuration on terminals they communicate with, you should define the terminals as roll screen devices in the TERMINAL statements, and enqueue them for static screen operation (ENQT/IOCB) when required. The exception is where a terminal is never used to communicate with the supervisor or system utilities (always used exclusively as a user static screen application terminal).

The only terminals that may be enqueued directly, by coding the label of the TERMINAL statement in the name operand of an ENQT statement, are the two special system terminals, \$SYSLOG and \$SYSPRTR. User-defined terminals and \$SYSLOGA are enqueued by coding the label of the TERMINAL statement in the name operand of an IOCB statement, and referencing the IOCB label in the ENQT name operand.



```

TERMPROG PROGRAM START
      .
      .
START ENQT
P1 PRINTEXT 'READY FOR INPUT'
      .
      .
P2 PRINTEXT T1
   DEQT
   PROGSTOP
T1 TEXT 'READY FOR INPUT'
   ENDPROG
   END

```

Figure 11-5. "msg" operand

In the PRINTEXT at P1 the storage area containing the EBCDIC character string READY FOR INPUT is implicitly generated (assembled) as part of the PRINTEXT instruction; the PRINTEXT at P2 references the user-defined (explicit) string at location T1.

Terminals are buffered devices. Data to be displayed on a terminal is transmitted to the terminal's buffer, and remains in the buffer until some condition occurs that forces the contents of the buffer to be displayed. Among the several buffer forcing conditions that can cause the contents of a buffer to be displayed or printed is the execution of a PRINTEXT with the LINE= or SKIP= forms control operands coded.

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

Figure 11-6. Forms control operands

The SPACES= forms control operand positions the message or cursor within a line, but does not force the device buffer. SKIP=, LINE=, and SPACES= may be coded as the only operand(s), or may be used with other operands, including msg. When coded with msg, the forms control operation is executed before the msg text is transmitted to the buffer.

In Figure 11-7, assume the loading terminal is \$SYSLOG, a 4979 Display. To better illustrate the effect of the forms control operands, the ENQT at START references an IOCB which sets NHIST= to 0. The entire screen will now operate as a roll screen work area.

```

TERMTEST PROGRAM START
START ENQT IOCB1
P1 PRINTTEXT LINE=0
P2 PRINTTEXT 'MESSAGE 1 ',SPACES=10,LINE=5
P3 PRINTTEXT 'MESSAGE 2 ',SPACES=20,SKIP=2
P4 PRINTTEXT 'MESSAGE 3 ',SPACES=70
P5 PRINTTEXT ' MESSAGE 4 ',SKIP=1
P6 PRINTTEXT 'MESSAGE 5 ',SPACES=5
P7 PRINTTEXT T1
P8 PRINTTEXT T2
P9 PRINTTEXT 'TEST ENDED',SKIP=1
DEQT
PROGSTOP
T1 TEXT 'MESSAGE 6 ',LENGTH=15
T2 TEXT 'MESSAGE 7 '
IOCB1 IOCB $SYSLOG,NHIST=0
ENDPROG
END

```

**Figure 11-7. PRINTTEXT example**

The PRINTTEXT at P1 illustrates a forms control operand coded without the msg command. Since the example is using a 4979 Display, this command readies the screen for display on line 0. If directed to a hardcopy device, this would be the equivalent of a page eject command.

The PRINTTEXT at P2 has both msg operand (text) and forms control operands coded. The forms control operation will be executed first. The LINE=5 forces the contents of the buffer onto line 0, and clears the buffer. (Because no msg operand was coded in the previous PRINTTEXT (P1), the buffer is empty, and nothing is displayed on line 0.) Next, the terminal is readied for display on line 5.

The SPACES=10 skips over the first ten buffer positions, and MESSAGE 1 goes in the next ten buffer positions (11 through 20). The text MESSAGE 1 is still in the buffer; no data has yet been displayed.

The PRINTTEXT at P3 performs the following functions:

1. The SKIP=2 forms control operand forces the buffer, displaying MESSAGE 1 on line 5.
2. The cursor is positioned for line 7 (SKIP=2), and the text MESSAGE 2 is placed in buffer positions 21 through 30, skipping over the first 20 buffer positions (SPACES=20).

After execution of the PRINTTEXT at P3, the display screen is as shown in Figure 11-8.

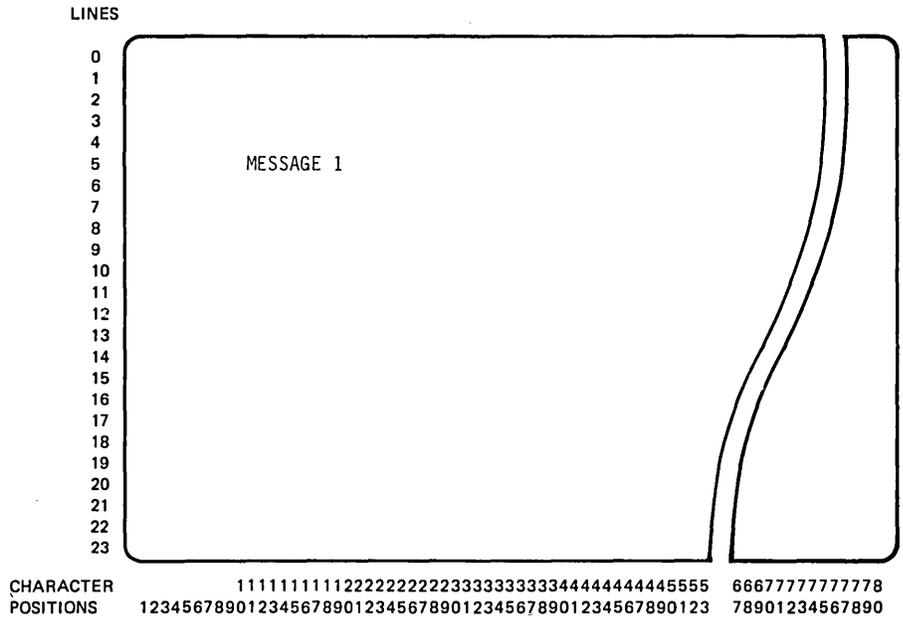


Figure 11-8. After P3 execution

The PRINTTEXT at P4 (Figure 11-7) has no LINE= or SKIP= operands coded, so the buffer is not forced out. The text MESSAGE 3 is concatenated to the current contents of the buffer, MESSAGE 2. MESSAGE 2 is in buffer positions 21 through 30. The SPACES=70 operand in the PRINTTEXT at P4 skips over 70 buffer positions, beginning with position 31. The text MESSAGE 3 will therefore occupy buffer positions 101 through 110.

The display screen is only 80 positions wide. Text data positioned outside the line length of a terminal is truncated, and therefore MESSAGE 3 will not be displayed. (OVFLINE=YES must be coded in the TERMINAL statement to allow display of text positioned outside the right margin.)

The PRINTTEXT at P5 (Figure 11-7) performs the following functions.

1. displays MESSAGE 2 in positions 21 through 30 on line 7 (SKIP=1 forces the buffer).
2. specifies line 8 for the next output line (SKIP=1) and places MESSAGE 4 in the first fifteen buffer positions. Figure 11-9 shows the screen after execution of the PRINTTEXT at P5.

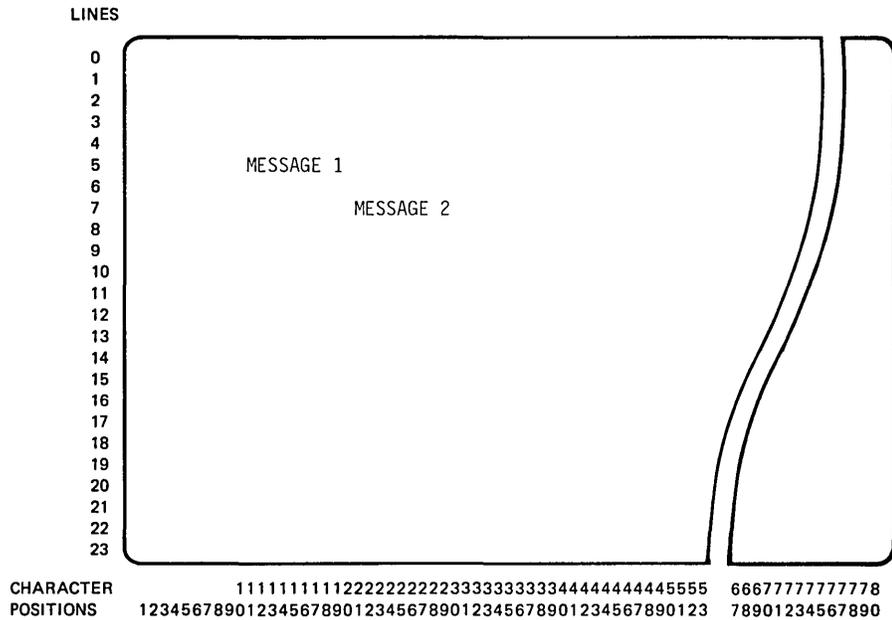


Figure 11-9. After P5 execution

The PRINTTEXT at P6 (Figure 11-7) skips buffer positions 16 through 20 (SPACES=5) and concatenates the text MESSAGE 5 into positions 21 through 30.

Explicitly defined text is also concatenated. The PRINTTEXT at P7 references the TEXT statement at T1. MESSAGE 6 is added to the buffer in positions 31 through 40. Although the text buffer at T1 is 15 characters long (LENGTH=15), only the data between the apostrophes is moved into the buffer. The PRINTTEXT at P8 adds MESSAGE 7 in positions 41 through 50.

When the PRINTTEXT at P9 executes, the buffer contents are displayed on line 8, and the cursor is moved to line 9 (SKIP=1). TEXT ENDED is placed in the first ten buffer positions. The screen now looks like Figure 11-10.

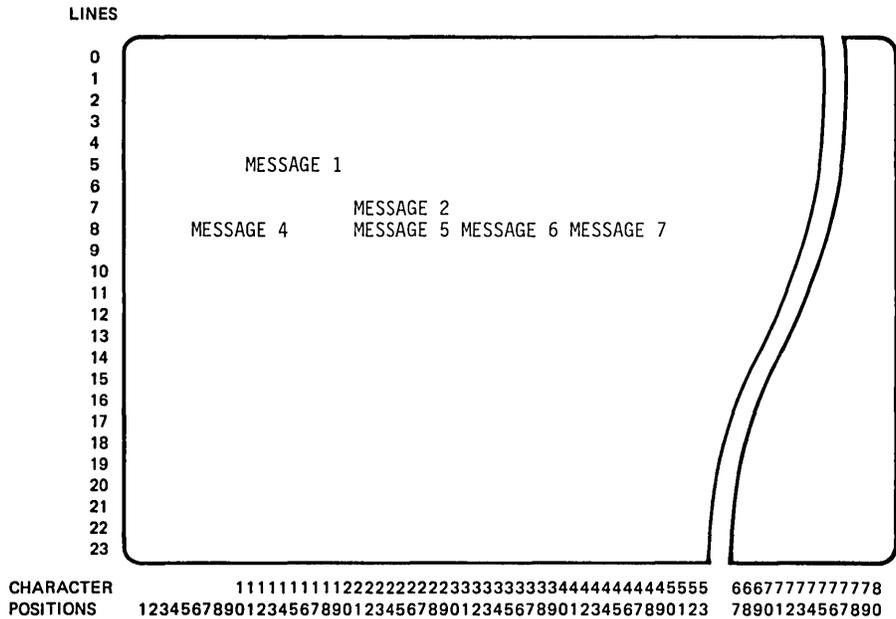


Figure 11-10. After P9 execution

There is no PRINTTEXT with a forms control operand following the PRINTTEXT at P9, but the TEST ENDED message will still be transferred from the buffer and displayed. Execution of a DEQT, like a LINE= or SKIP= forms operation, is a buffer-forcing condition.

In the example in Figure 11-7, the program would still execute correctly if the DEQT were not coded. The PROGSTOP statement will dequeue the terminal (implicit DEQT) and force the buffer. You should still get in the habit of coding explicit DEQTs, because the system cannot be relied upon to perform such housekeeping chores in all cases. For example, if the terminal instructions in Figure 11-7 were part of a secondary task and the DEQT were left out, the terminal would remain enqueued and unavailable to the rest of the system after the secondary task completed execution. Unlike the PROGSTOP, execution of an ENDTASK instruction does not automatically issue a DEQT.

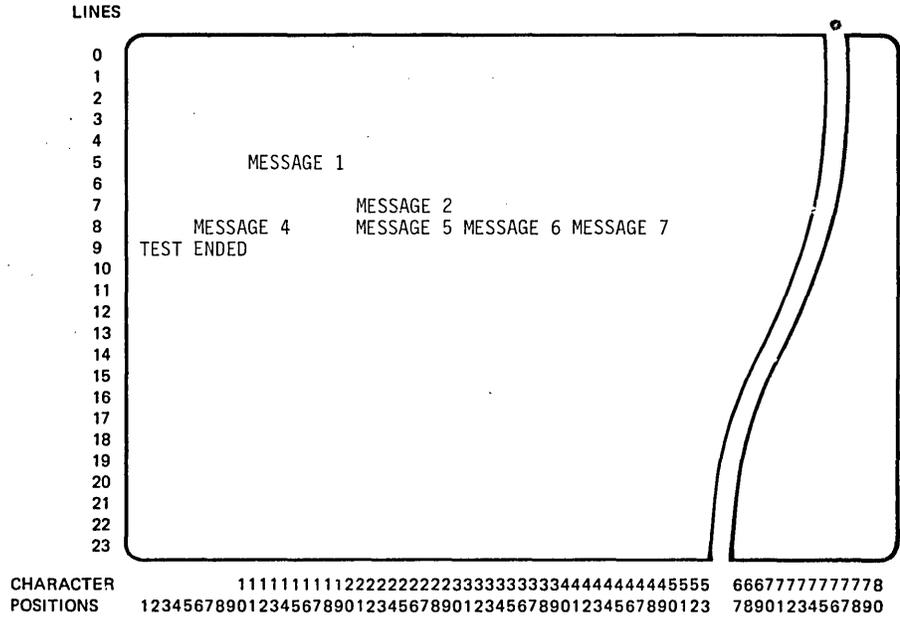


Figure 11-11. After P1 through DEQT

Figure 11-11 shows the screen after all PRINTTEXT instructions and the DEQT have been executed.

When writing to roll screen devices, an at sign (@) imbedded in the text will be interpreted as a new line or "carriage return" control character. In Figure 11-12, the programs T1 and T2 are logically equivalent.

T1	PROGRAM	S1		T2	PROGRAM	S2
S1	ENQT			S2	ENQT	
P1	PRINTTEXT	'FIRST MSG'		X1	PRINTTEXT	'FIRST MSG'
P2	PRINTTEXT	'2ND MSG',SKIP=1		X2	PRINTTEXT	'@2ND MSG'
D	DEQT			X	DEQT	
	PROGSTOP				PROGSTOP	
	ENDPROG				ENDPROG	
	END				END	

Figure 11-12. @ operation

The PRINTTEXT statements at P1 and X1 are identical, and will put the text FIRST MSG in the buffer. In program T1, the SKIP=1 operand in the PRINTTEXT at P2 will force the buffer, displaying FIRST MSG on the current line, and move the display position to the next line. 2ND MSG will be placed in the buffer.

The @ imbedded in the msg operand of the PRINTTEXT at X2 (program T2) has the same effect as SKIP=1, forcing the buffer contents onto the current line, and moving the display position to the next line. Unlike the SKIP= and LINE= operands, the @ or new line operation is executed at the time it is encountered in the character buffer. The SKIP=1 operand in task T1 executes before 2ND MSG is transferred to the buffer, because SKIP= and LINE= operations always execute before the buffer transfer. The new line operation in task T2 is also executed before 2ND MSG is transferred to the buffer because the @ precedes the 2ND MSG text. Were the @ imbedded further along in the text string, characters to the left of the @ would be concatenated to the FIRST MSG text and displayed on the same line as FIRST MSG, while characters to the right of @ (as shown in Figure 11-12) would be displayed on the next line.

In both T1 and T2, the 2ND MSG text is moved out of the buffer and displayed by execution of the DEQT (D or X).

```
label PRINTTEXT msg,SKIP=,LINE=,SPACES=,MODE=,PROTECT=
```

Figure 11-13. MODE= operand

When you want the @ character to act as a normal text character (not to be interpreted as a new line character), the MODE= keyword operand should be coded as MODE=LINE.

The MODE= operand has a special function when used with PRINTTEXT instructions directed to static screen devices (4978s or 4979s) with protected data areas.

```
label PRINTTEXT msg,SKIP=,LINE=,SPACES=,MODE=,PROTECT=
```

Figure 11-14. PROTECT= operand

Protected data is written to a static screen by coding the PROTECT= keyword operand as PROTECT=YES. If MODE=LINE is coded in a subsequent PRINTTEXT that is writing to a line containing protected data, the protected areas are automatically skipped over when the buffer is transferred to the screen.

## READTEXT INSTRUCTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "READTEXT."

The READTEXT instruction is used to read an alphanumeric text string, entered by a terminal operator, into a user-defined text buffer in storage.

```

label  READTEXT  loc, pmsg, PROMPT=, MODE=, SKIP=, LINE=, SPACES=
OPTIONAL  MUST BE CODED  OPTIONAL
  
```

Figure 11-16. READTEXT format

The loc operand is the label of the first location of the storage area that will receive the EBCDIC character string from the terminal. The READTEXT instruction (also PRINTTEXT) operates with TEXT statements, using the length and count control bytes that precede a character buffer generated by a TEXT statement assembly. The loc operand is, therefore, usually the label of a TEXT statement; if it is coded as the label of a character buffer not generated by a TEXT statement, the user must set up the control bytes preceding the buffer to meet TEXT statement conventions.

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

Figure 11-17. pmsg and PROMPT= operands

The pmsg operand is the prompt message (enclosed in apostrophes) or the label of a TEXT statement containing the prompt message you wish displayed before pausing to accept the operator input. The pmsg operand works in conjunction with the PROMPT= keyword operand. If PROMPT= is coded as PROMPT=UNCOND (which is the default if it is not coded), the prompt message specified by the pmsg operand will always be written. If PROMPT= is coded as PROMPT=COND, advance input is allowed, and the prompt message may or may not be written. Advance input allows an operator to enter more information on a line than is suggested by the prompt message for that line. An operator familiar with a certain prompt/response sequence can enter all items in response to the first prompt, thereby skipping succeeding prompt messages. The use of PROMPT=COND will be illustrated in an example later in this section.

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

Figure 11-18. MODE= operand

The MODE= operand may be coded MODE=WORD (the default, if not coded) or MODE=LINE. If MODE=WORD is coded, transfer of data from a terminal buffer to a user text buffer is terminated by:

1. a blank (space) character in the input field
2. exhaustion of the character count in the user text buffer (input exceeding input buffer length — truncation of input occurs)
3. if directed to a static screen, the beginning of a protected field.

If MODE=LINE is coded, the input data may contain imbedded blanks without terminating the transfer. If a READTEXT with MODE=LINE is directed to a static screen, protected areas do not occupy user TEXT buffer positions; only the unprotected areas are read.

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

Figure 11-19. Forms control operands

The SKIP=, LINE=, and SPACES= operands perform the same function as with the PRINTTEXT instruction, specifying the line and position within the line where the next operation will take place.

READTEXT operation, including some of the operand variations just discussed, is illustrated in Figure 11-20. Assuming the program is loaded from a 4979 Display, the ENQT at START changes the (defaulted) history area from 12 lines to none, and enqueues the terminal. The LINE=3 operand in the READTEXT at R1 readies the terminal for display on line 3, and the loc operand specifies a 20-character text buffer at location T1 as the storage area that will receive the input data.

The READTEXT at R2 specifies T2 as the input buffer. The pmsg operand is the label of the TEXT statement T3, containing the prompt message ENTER PART NBR:.

When the READTEXT at R1 executes, the prompt message ENTER PART NAME will be displayed on line 3, the cursor will be positioned just following the colon in the prompt message, and task TERM will be suspended, waiting for operator input.

As an operator keys an entry onto the screen, there is no program involvement. The actual input operation (transfer of terminal buffer information to storage) does not begin until the program is signalled that the input is complete. When the operator is satisfied that the input is correct, he/she will press the ENTER key, initiating the actual transfer. (The Program Function keys are also interrupt generating, and are frequently used in operator/terminal communication. They will be covered later in this section.)

Assume that the operator, in response to the ENTER PART NAME: prompt, enters BRACKETS, and then presses the ENTER key. The READTEXT at R1 will transfer the contents of the terminal buffer to the text buffer at T1. The READTEXT at R2 will then display the prompt message ENTER PART NBR: on the next line, and TERM will again be suspended, waiting for operator input.

The operator then enters 105636, and presses ENTER again. The READTEXT at R2 transfers 105636 to the text buffer at T2, and the program runs to completion.

```

TERM      PROGRAM      START
IOCB1     IOCB         NHIST=0
START     ENQT         IOCB1
R1        READTEXT    T1,'ENTER PART NAME:',LINE=3
R2        READTEXT    T2,T3,PROMPT=COND
          DEQT
          PROGSTOP
T1        TEXT         LENGTH=20
T2        TEXT         LENGTH=6
T3        TEXT         'ENTER PART NBR:'
          ENDPROG
          END

```

Figure 11-20. READTEXT operation

If the operator knows that the prompt ENTER PART NBR: will follow the first prompt of ENTER PART NAME:, he may make both the part name and part number entries on the same line (line 3), in response to the first prompt. The READTEXT at R2 has PROMPT=COND coded, meaning that the prompt message ENTER PART NBR: will be issued conditional on the absence of advance input in the previous operation.

If the operator entered BRACKETS 105636 when the first prompt ENTER PART NAME: was displayed, the READTEXT at R2 would detect advance input, and would transfer the second part of the entry (the advance input, 105636) into the text buffer at T2, without issuing the prompt message ENTER PART NBR:, and without suspending TERM to wait for the ENTER key.

The presence of advance input is indicated by an imbedded blank within an input character string. PROMPT=COND will, therefore, not work if the previous operation (the operation where advance input is expected) has MODE=LINE in effect, allowing imbedded blanks. In this case, the operation would not terminate when a blank in the input is found.

Since advance input (PROMPT=COND) can only be used when MODE=WORD is also used, care must be taken that no blanks, other than those separating entries, appear in the input string. For example, if the operator wished to use advance input, but mistakenly entered WALL BRACKETS 105636, the first input operation (READTEXT at R1) would terminate with the blank between WALL and BRACKETS, and WALL would be transferred to the text buffer T1. The READTEXT at R2, operating with advance input because of the imbedded blank, would transfer BRACKE into text buffer T2, would not issue the prompt at T3, and would terminate due to exhaustion of the character count of 6 in the input buffer. The actual part number 105636 would never be read.

## OPERATOR CONTROL OF PROGRAM EXECUTION

### PF and Attention Key Handling

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "Terminal I/O – Attention Handling", "ATTNLIST", "ENDATTN."

Attention routines are user routines that service interrupts generated by pressing the ATTENTION key on a terminal (review *Attention Lists* in Section 3). The ATTNLIST statement is used to define operator entries and corresponding program locations that will receive control when the defined entries are made.

The Program Function keys on 4978/4979/3101 M2 Displays generate interrupts similar to those generated by the ATTENTION key and the entry points of routines to service these PF interrupts may also be defined using the ATTNLIST statement.

The ATTNLIST statement in Figure 11-21 defines three attention routine entry points. SET1, the first entry point, operates with the ATTENTION key. If an operator presses ATTENTION, enters the characters ONE, and then presses the ENTER key, location SET1 receives control.

```

PROG      PROGRAM      START
ATTNLIST (ONE,SET1,$PF1,P1,$PF,END)
START    IF            (SWITCH,EQ,1),GOTO,PRINT
          IF            (SWITCH,EQ,2),GOTO,PFPRINT
          IF            (SWITCH,EQ,3),GOTO,OUT
          .
          .
          .
BACK     GOTO          START
PRINT    MOVE          SWITCH,0
          PRINTTEXT    'ATTENTION INTERRUPT'
          PRINTTEXT    SKIP=1
          GOTO          START
PFPRINT  MOVE          SWITCH,0
          PRINTTEXT    'PROGRAM FUNCTION KEY #1'
          PRINTTEXT    SKIP=1
          GOTO          START
SET1     MOVE          SWITCH,1
          ENDATTN
P1       MOVE          SWITCH,2
          ENDATTN
END      MOVE          SWITCH,3
          ENDATTN
OUT      PROGSTOP
SWITCH  DATA          F'0'
          ENDPROG
          END

```

Figure 11-21. Attention routines

Program Function keys are identified in an ATTNLIST statement by the system convention "\$PFx", where x is an integer between 1 and 6, corresponding to Program Function keys PF1 through PF6. In this example, location P1 will get control when PF1 is pressed. (The x = integer between 1 and 6 applies to the 4979 Display. When using the 4978 Display, many more interrupting keys are available, and the PFx in an ATTNLIST statement may range between PF1 and PF254.) The 3101 M2 has 8 program function keys available.

When \$PF is used without a specific number, it is interpreted as all PF keys not previously defined (to the left of this entry) in this ATTNLIST statement. In Figure 11-21, Program Function key 1 is previously defined (middle operand pair \$PF1,P1), so location END will get control if PF2 through PF6 is pressed, and P1 will get control if PF1 is pressed. If the second and third operand pairs in the ATTNLIST statement were coded in reverse order, END would get control when any PF key was pressed, including PF1; control would never be transferred to P1.

Attention routines execute as part of the system keyboard task, not as part of the user task within which they appear. Since user interference with system keyboard task execution is clearly undesirable, certain I/O and task control instructions are not allowed within attention routines. See the reading assignment for a list of excluded instructions.

When the keyboard task detects an ATTENTION or PF key interrupt for a task with the appropriate entry points defined in an ATTNLIST statement, part of the response process is to briefly enqueue the interrupting terminal (ENQT). If the user task has an ENQT already in effect, the keyboard task is prevented from getting in. For an interrupt resulting from the operator's pressing the ATTN key, the system cannot present the > prompt character until the user program issues a DEQT, at which time the > will be displayed. For interrupts generated by depression of PF keys or the ENTER key (while the terminal is enqueued by the user), the system returns an identifying code to the user program. This code can be examined by user instructions to determine which key was pressed. All PF keys and the ENTER key will present identifying codes; the user is not restricted to those PF keys defined in an ATTNLIST statement whose function has been temporarily inhibited by a user ENQT. Examples later in this section will illustrate how to retrieve and use the identification codes resulting from PF key or ENTER key interrupts.

Attention routines execute on hardware level 1, thereby automatically preempting execution of all user tasks on levels 2 and 3. They should, therefore, be kept very short and are usually limited to the setting of a program switch (or posting an ECB) which is checked during normal program execution. The example in Figure 11-21 illustrates this.

This program checks a program indicator for a value, and branches to different program locations, depending on what value is found. In this case, the indicator is the word at location SWITCH, which has an initial value of zero. As long as SWITCH remains zero, the program will loop between START and BACK.

Pressing the ATTENTION key and entering ONE results in execution of the attention routine at SET1, altering the value of SWITCH to = 1. When the IF statement at START is next executed, control will be transferred to PRINT, and the message ATTENTION INTERRUPT will be displayed. Pressing PF1 will set SWITCH=2 (attention routine at P1), and result in a transfer to PFPRINT, which will display PROGRAM FUNCTION KEY #1. Pressing any Program Function key other than PF1 will end the program (SWITCH=3, transfer to location OUT). Note that the attention routine at location END (PF2 through PF6) only sets location SWITCH to cause a later transfer to the PROGSTOP; PROGSTOP is one of the instructions excluded from attention routines, and cannot be issued from within the attention routine itself.

## QUESTION Instruction

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "QUESTION."

The QUESTION statement provides another way of altering program execution through terminal input. QUESTION displays a prompt message, usually in the form of a question, and branches to a specified location based on the response entered on the terminal.

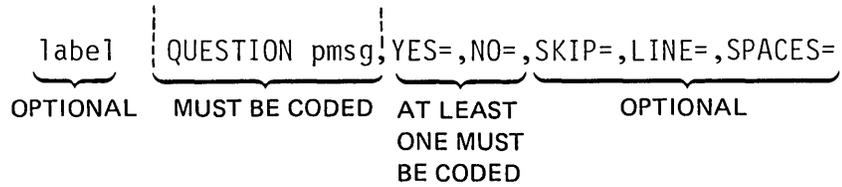


Figure 11-22. QUESTION format

The `pmsg` operand is coded as the prompt message, contained within apostrophes, or as the label of a `TEXT` statement containing the prompt message.

The `YES=` and `NO=` operands are coded with the labels of the program locations which are to get control if a `YES` or a `NO` response is entered. The only valid responses to a `QUESTION` prompt are `Y` and `N` (or any character string beginning with `Y` or `N`). Either `YES=` or `NO=` may be left uncoded, but not both. Entering the uncoded response will result in transfer to the instruction following the `QUESTION` statement.

## WAIT KEY Instruction

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "WAIT."

In addition to the implied wait for operator input that is provided by the `READTEXT` and `QUESTION` instructions, the user can wait for the `ENTER` key or `PF` keys at any time, using a special variation of the `WAIT` statement, `WAIT KEY`. This instruction suspends the issuing task until the `ENTER` key or one of the `PF` keys is pressed, at which time the `WAIT` terminates, and execution continues with the instruction following the `WAIT KEY`. There is no automatic transfer to an attention routine; execution of a `WAIT KEY` instruction enqueues the terminal and temporarily inhibits the `ATTNLIST` capability during the time the task is suspended due to that `WAIT` instruction, just as the `ATTNLIST` function is inhibited while an `ENQT` is in effect.

`WAIT KEY` is most often used by tasks operating terminals as static screen devices. In the roll screen examples shown before, issuing a `READTEXT` command caused a suspension of the issuing task, waiting on operator input. Execution resumed, and the input operation completed only when the operator signalled the program that the input data was available by pressing the `ENTER` key.

When operating with static screens, the `ENTER` key signals that an entire page (screen) of input data is available. `READTEXT` instructions directed to a static screen terminal therefore do not cause the issuing task to wait; the input data is expected to be present, and is transferred immediately.

`WAIT KEY` allows a task with a terminal enqueued as a static screen device to wait on the `ENTER` key (or `PF` keys), even though the implied wait with `READTEXT` is not operative.

*Note:* When operating with static screen devices, the implied wait with READTEXT is inoperative only when the READTEXT has no prompt message coded. Terminal input operations that are obviously intended for operator dialogue, such as a READTEXT with the pmsg operand coded, or a QUESTION instruction, still work the same as with roll screens, automatically suspending the issuing task.

As already noted, the ATTNLIST capability is inhibited when a terminal is enqueued by a task as either a roll screen or static screen device, and/or when the task is suspended by a WAIT KEY instruction. Although automatic transfer to individual attention routine entry points associated with specific PF keys is no longer possible, the user can find out which key was pressed, and do the routing personally. An integer value equal to the numeric designation of the PF key is passed back to the user task in the second word of the task's TCB (taskname+2), and may be examined by the user program. The code passed back for the ENTER key is zero. For PF1, taskname+2 will contain a 1, for PF2 a 2, and so on through 6 for PF6. The code can be checked, and a transfer decision made, using IF statements or a computed GOTO.

*(Note:* When using the 4978 Display, many more interrupting keys and corresponding identification codes are available than with the 4979 terminal discussed above. See the topic "\$PFMAP" in *Section 14. Utility Programs* for an aid in determining the identification codes associated with particular 4978 interrupting keys.)

## HARDCOPY PF Key

One of the operands in the TERMINAL statement defining 4978/4979 Displays is HDCOPY=. This is coded with the symbolic name of a hardcopy terminal and a PF key number, in the format HDCOPY=(termname,keynbr). The termname must be coded. If keynbr is not coded, it defaults to 6, indicating Program Function key PF6.

Whenever the PF key specified in the HDCOPY= operand is depressed, the present screen contents are printed out on the designated hardcopy device. The default for the 4979 supported by the supplied supervisor is HDCOPY=(\$SYSPRTR,6), causing the screen contents to be printed on the 4974 Matrix Printer whenever PF6 is depressed.

Not knowing which PF key you may designate to activate the hardcopy system function, all examples in this section address Program Function keys PF1 through PF6 (as though HDCOPY= were coded HDCOPY=(\$SYSPRTR,6)).

In coding your own programs, you should be aware that the key you specify in the HDCOPY= operand is not available to you for other purposes. If specified in an ATTNLIST statement, the associated entry point will never receive control nor will pressing the hardcopy PF key terminate a WAIT KEY operation, or present its code in taskname+2.

## STATIC SCREEN CODING EXAMPLE

In the following several illustrations (Figures 11-23 through 11-43), a simple static screen program is developed, using most of the terminal instructions already discussed, and introducing some new instructions applicable only to static screen operation.

The initial portion of this program operates the terminal as a roll screen device, with NHIST=0. The rest of the program uses the terminal in the static screen mode. An IOCB will be required for each of the two modes.

Operator instructions are displayed requiring the operator to (1) end the program, or (2) bring up the entry screen (static screen) and proceed. The operator's decision is communicated to the program using the ATTNLIST facility, so an ATTNLIST statement will also be required.

Figure 11-23 shows the two IOCBs, the ATTNLIST statement, and the associated attention routines.

```
XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIC
ATTNLIST (END,OUT,$PF,STATIC)
.
.
.
OUT POST ATTNECB,1
ENDATTN
STATIC POST ATTNECB,-1
ENDATTN
ATTNECB ECB
.
.
.
ENDPROG
END
```

Figure 11-23. IOCB/ATTNLIST

Figure 11-24 is the entire roll screen portion of the program. Execution begins at location START, with the ENQT directed to IOCB1. The IOCB changes NHIST=12 to NHIST=0 for the loading terminal (no terminal name specified in the IOCB, default to loading terminal, and assuming loading terminal is a 4979 with NHIST=12 normally in effect).

Now that the loading terminal is enqueued, the five PRINTTEXT statements following the ENQT display the program title and operator directions on the screen. Since operator control has been defined through an ATTNLIST, and ATTNLIST is inhibited while the terminal is enqueued, the last PRINTTEXT is followed by a DEQT, placing the ATTNLIST in effect.

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIC
ATTNLIST (END,OUT,$PF,STATIC)
START ENQT IOCB1
PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
PRINTTEXT ' THE PROGRAM'
PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
PRINTTEXT ' BRING UP THE ENTRY SCREEN'
DEQT
CHECK WAIT ATTNECB,RESET
IF (ATTNECB,EQ,1),GOTO,ENDIT
ENTRY ENQT IOCB2
.
.
.
ENDIT PROGSTOP
.
.
.
OUT POST ATTNECB,1
ENDATTN
STATIC POST ATTNECB,-1
ENDATTN
ATTNECB ECB
.
.
.
ENDPROG
END

```

Figure 11-24. Roll screen portion

The ECB at location ATTNECB assembles with an initial value in the first word of -1 indicating "event complete". The WAIT at location CHECK is coded with a RESET operand, which resets the first word of the ECB at ATTNECB to zero before the WAIT is executed. A zero in the first word of an ECB indicates "event not occurred," so the WAIT at CHECK will suspend task XMPLSTAT, waiting on event ATTNECB. If the WAIT has been coded without the RESET operand, the WAIT would have executed as a no-op.

If the operator presses ATTENTION, enters END and presses RETURN, the attention routine at OUT will execute, posting the ECB at ATTNECB with a +1 (first word = 1). A value other than zero in the first word of the ECB indicates "event complete," and the WAIT operation terminates. Execution continues with the IF statement following the WAIT, which will transfer control to location ENDIT.

If the operator wants to proceed with the CLASS ROSTER PROGRAM and presses a PF key, ATTNECB will be posted with a value of -1 by the attention routine at STATIC. The WAIT will terminate, the IF that follows will not transfer control to ENDIT (ATTNEBC NOT = +1), and execution will continue with the ENQT at location ENTRY, which is the beginning of the static screen portion of the program.

After the program title and operator instructions have been written to the terminal (while the program is waiting at CHECK for the operator response), the screen looks like Figure 11-25.

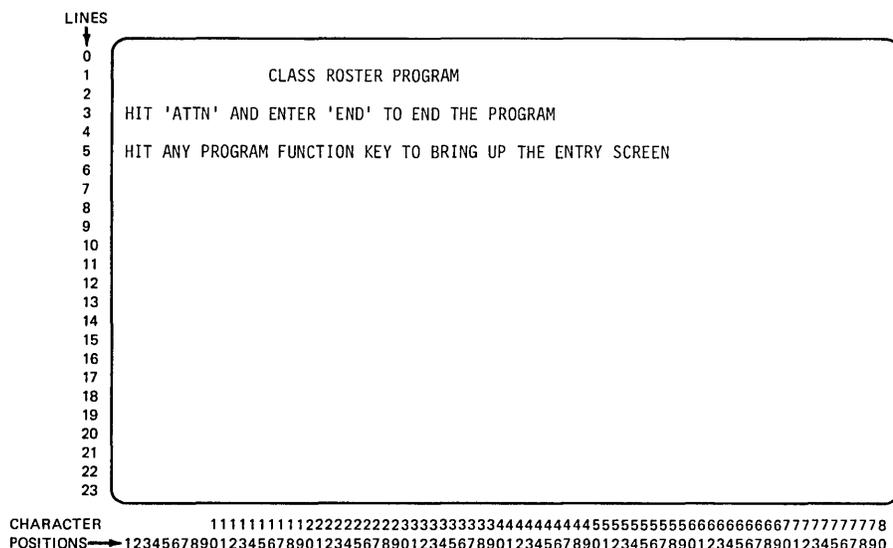


Figure 11-25. Initial operator instructions

Assuming the operator pressed a PF key, execution now continues at location ENTRY (Figure 11-26). The ENQT enqueues the terminal as a static screen device.

## ERASE Instruction

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "ERASE."

An automatic erase of a roll screen is performed by the system each time the page size of the screen is exceeded. Erasure of a static screen device is a user responsibility, and the ERASE instruction is, therefore, valid only for static screens.

You can select how much you want to erase, from as little as a single character position to the entire screen. In Figure 11-26, the ERASE following the ENQT will erase the entire screen. The MODE= operand defines the ending point of the erase operation; in this case, the end of the screen. The starting point of the erase is determined by SKIP=, LINE=, and SPACES= forms operands, in this example defaulting to LINE=0, SPACES=0. TYPE= specifies whether only unprotected data should be erased (TYPE=DATA) or if the erase applies to protected data also (TYPE=ALL).

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIC JUNCTION KEY TO',SKIP=Z
ATTNLIST (END,OUT $* E ENTRY SCREEN'
START ENOT
CHECK IF ATTNECB,RESET
IF (ATTNECB,EQ,1),GOTO,ENDIT
ENTRY ENQT IOCB2
ERASE MODE=SCREEN,TYPE=ALL
TERMCTRL BLANK
PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
PRINTTEXT ' PF1 = DELETE ENTRY 1'
PRINTTEXT ' PF2 = DELETE ENTRY 2'
PRINTTEXT 'PF3 = DELETE ENTRY 3',SKIP=1
PRINTTEXT 'PF4 = DELETE ENTRY 4'
.
.
ENDPROG
END

```

Figure 11-26. Operator directions

## TERMCTRL Instruction

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "TERMCTRL."

TERMCTRL is used for several specialized functions, most of which are device/hardware feature dependent control operations. The 3101 operating in block mode uses an attribute byte (prefixed to the data field) to define the display mode as low or high intensity, blinking or non-display. The TERMCTRL statement is used to set the characteristics of the attribute byte. In Figure 11-26, the TERMCTRL BLANK instruction blanks the 4979 display screen.

The remainder of this portion of the program is going to format the display screen by executing a series of PRINTTEXT instructions. When several operations are performed sequentially, the 4979 screen exhibits a flickering that some people find annoying. Issuing the TERMCTRL BLANK turns off the display capability of the screen, allowing the series of output operations to take place without visible flicker. After the formatting has been completed, another TERMCTRL function will be used to display the finished screen.

The five PRINTTEXT instructions following the TERMCTRL will write some operator guides at the top of the screen. When these instructions have executed, the screen would look like Figure 11-27 (assuming an unblanked screen).

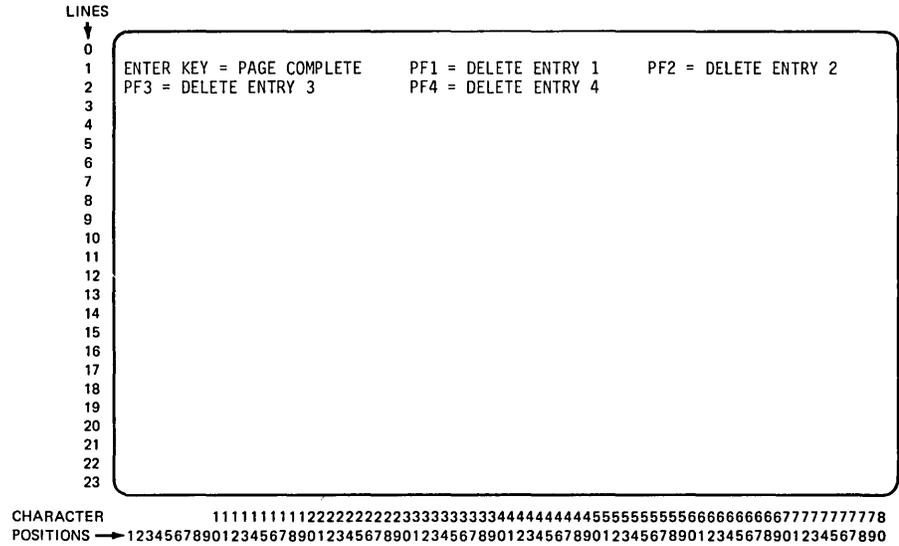


Figure 11-27. Operator directions/screen

In Figure 11-28, execution continues with the PRINTTEXT at location HDR. This instruction writes a screen-wide (80 character) line of hyphens, separating the operator guide area just written from the rest of the screen. The text buffer referenced by this instruction (location DASHES) is not the label of a TEXT statement, but is a user-defined text buffer. Since PRINTTEXT uses the control bytes that precede text buffers generated by TEXT statements, the user must code the control bytes when defining non-TEXT statement text buffers.

The DATA statement preceding location DASHES is coded as X'5050', establishing a length byte of 80 and a count byte of 80 (hex 50=decimal 80). This tells the PRINTTEXT at HDR that the buffer is 80 character positions long, and that all 80 positions contain data.

```

XEMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=0 PAGE COMPLETE',LINE=1
ATTNLIST (END PF1 = DELETE ENTRY 1'
PF2 = DELETE ENTRY 2'
PF3 = DELETE ENTRY 3 ',SKIP=1
PF4 = DELETE ENTRY 4'
HDR PRINTTEXT DASHES,PROTECT=YES,LINE=3
PRINTTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
PRINTTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
PRINTTEXT DASHES,PROTECT=YES,LINE=5
MOVE LINENBR,6
.
.
.
DATA X'5050'
DASHES DATA 80C'- '
.
.
ENDPROG
END

```

Figure 11-28. Non-standard text buffer

The PROTECT=YES operand specifies that the line of hyphens be written as protected data. Protected data cannot be altered by operator input.

The next PRINTTEXT places CLASS NAME: in the first eleven positions of line 4, and the following one puts INSTRUCTOR NAME: on the same line, with both messages protected.

The last PRINTTEXT in Figure 11-28 writes another separator line of hyphens, again using the user-defined text buffer at DASHES. Figure 11-29 shows how the screen would look if it were displayed at this point.

```

LINES
↓
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3
4 -----
5 CLASS NAME:                    INSTRUCTOR NAME:
6 -----
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
CHARACTER      1111111111222222222233333333334444444444555555555566666666667777777778
POSITIONS → 1234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-29. Header

The rest of the screen formatting section of the program is shown in Figure 11-30. This portion will format the remainder of the screen into four data entry areas.

First, the variable LINENBR is set to 6. Next, a DO loop is defined, specifying four executions of the loop, corresponding to the four data entry areas to be formatted.

All PRINTTEXT instructions within the loop have the LINE= operand coded, with the variable name LINENBR, rather than as an integer constant. Before this first execution of the DO loop, LINENBR was initialized to 6. The first PRINTTEXT writes the protected characters NAME: into the first 5 positions of line 6, and the second PRINTTEXT leaves 25 unprotected spaces following NAME:, and writes STREET: to the same line.

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=ST
ATTNLIST (FND
HDR
PRINTTEXT DASHES,PROTECT=YES,LINE=3
PRINTTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
PRINTTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
PRINTTEXT DASHES,PROTECT=YES,LINE=5
MOVE LINENBR,6
DO 4,TIMES
PRINTTEXT 'NAME:',LINE=LINENBR,PROTECT=YES
PRINTTEXT 'STREET:',LINE=LINENBR,SPACES=30,PROTECT=YES
A1 ADD LINENBR,1
PRINTTEXT 'CITY:',LINE=LINENBR,SPACES=30,PROTECT=YES
A2 ADD LINENBR,1
PRINTTEXT 'STATE:',LINE=LINENBR,SPACES=30,PROTECT=YES
ADD LINENBR,3
ENDDO
PRINTTEXT LINE=4,SPACES=11
TERMCTRL DISPLAY
WAITONE WAIT KEY
.
.
.
LINENBR DATA F'0'
ENDPROG
END

```

Figure 11-30. Finish formatting the screen

Next, the ADD at A1 increases LINENBR by 1, and the PRINTTEXT that follows is directed to line 7, LINENBR is again incremented (ADD at A2), and the last PRINTTEXT is directed to line 8. The ADD just preceding the ENDDO increases LINENBR by 3, skipping down to the next data entry area to be formatted.

After four executions of the DO loop, the PRINTTEXT immediately following the ENDDO statement is executed. This PRINTTEXT positions the cursor just to the right of the CLASS NAME: message in the screen header, above the four data entry areas just formatted in the DO loop. The TERMCTRL DISPLAY command removes the blanking from the screen, and displays the cursor at the position determined by the previous PRINTTEXT. Figure 11-31 shows the fully formatted screen that is now displayed.

LINES

```
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3 -----
4 CLASS NAME:  ---                INSTRUCTOR NAME:
5 -----
6 NAME:                               STREET:
7                                     CITY :
8                                     STATE :
9
10
11 NAME:                               STREET:
12                                    CITY :
13                                    STATE :
14
15
16 NAME:                               STREET:
17                                    CITY :
18                                    STATE :
19
20
21 NAME:                               STREET:
22                                    CITY :
23                                    STATE :
```

```
CHARACTER      1111111111222222222233333333334444444444555555555566666666677777777778
POSITIONS     12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Figure 11-31. Completed format

The program is in a wait state, suspended by execution of the WAIT KEY at location WAITONE. The program will not be activated again until the operator presses the ENTER key or one of the PF keys.

The screen is now completely formatted, and ready for data entry. Figure 11-32 shows the complete screen formatting portion of the program.

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIC
.
.
.
ENTRY ENQT IOCB2
ERASE MODE=SCREEN,TYPE=ALL
TERMCTRL BLANK
PRINTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
PRINTEXT ' PF1 = DELETE ENTRY 1'
PRINTEXT ' PF2 = DELETE ENTRY 2'
PRINTEXT 'PF3 = DELETE ENTRY 3 ',SKIP=1
PRINTEXT 'PF4 = DELETE ENTRY 4'
HDR PRINTEXT DASHES,PROTECT=YES,LINE=3
PRINTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
PRINTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
PRINTEXT DASHES,PROTECT=YES,LINE=5
MOVE LINENBR,6
DO 4,TIMES
PRINTEXT 'NAME:',LINE=LINENBR,PROTECT=YES
PRINTEXT 'STREET:',LINE=LINENBR,SPACES=30,PROTECT=YES
A1 ADD LINENBR,1
PRINTEXT 'CITY :',LINE=LINENBR,SPACES=30,PROTECT=YES
A2 ADD LINENBR,1
PRINTEXT 'STATE :',LINE=LINENBR,SPACES=30,PROTECT=YES
ADD LINENBR,3
ENDDO
PRINTEXT LINE=4,SPACES=11
TERMCTRL DISPLAY
WAITONE WAIT KEY
.
.
.
DASHES DATA X'5050'
DATA 80C'- '
.
.
.
LINENBR DATA F'0'
ENDPROG
END

```

Figure 11-32. Screen formatting section

The operator may position the cursor at will, and enter data in any unprotected area of the screen. Positioning the cursor at LINE=4, SPACES=11 (PRINTEXT following ENDDO), is a convenience to the operator, not a required function — the operator could have used the cursor positioning keys to move the cursor to the same position.

Some cursor-positioning functions are automatically provided by the hardware. Assume that the operator enters SERIES/1 HARDWARE in the space immediately following the protected CLASS NAME:

message, and then presses the tab right key (). The cursor

will automatically skip over the protected INSTRUCTOR NAME: field, and position itself at the beginning of the unprotected area which follows, as shown in Figure 11-33.

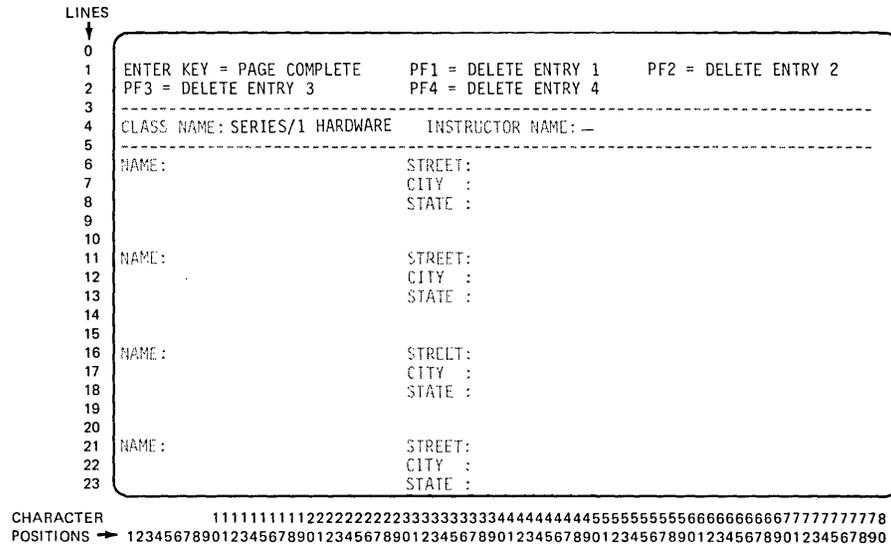


Figure 11-33. Cursor movement (1)

After entering the instructor name, the next tab right key depression results in the cursor position shown in Figure 11-34, ready for the first student name entry.

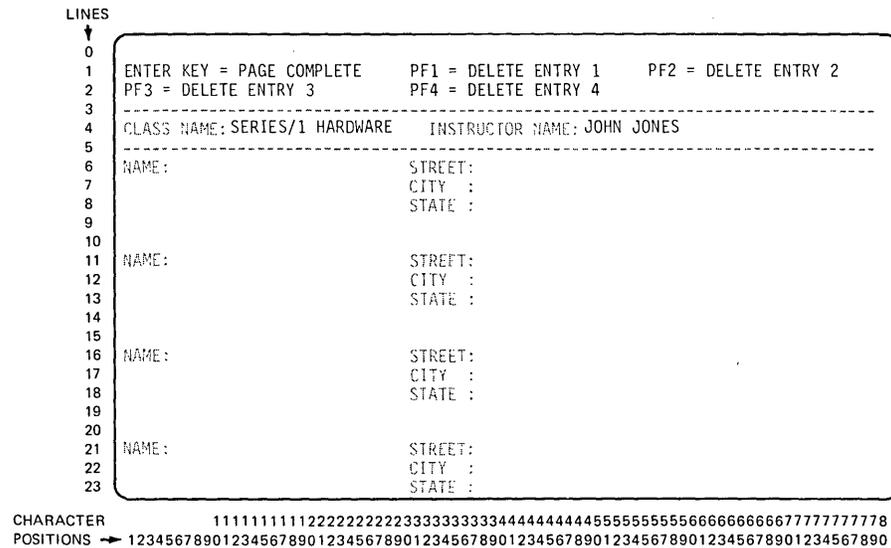


Figure 11-34. Cursor movement (2)

Each successive tab key depression results in an automatic skip of the cursor to the beginning of the next unprotected area on the screen. In this example, the cursor will successively tab to NAME:, STREET:, CITY:, and STATE:, and then down to the NAME: in the next data entry area, as shown in Figure 11-35.

```

LINES
↓
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3
4 CLASS NAME: SERIES/1 HARDWARE  INSTRUCTOR NAME: JOHN JONES
5
6 NAME: JOHN JAMES                STREET: 111 GRANT AVENUE
7                                CITY : ENDICOTT
8                                STATE : NEW YORK 13760
9
10
11 NAME: —                        STREET:
12                                CITY :
13                                STATE :
14
15
16 NAME:                          STREET:
17                                CITY :
18                                STATE :
19
20
21 NAME:                          STREET:
22                                CITY :
23                                STATE :
CHARACTER      11111111112222222222333333333344444444445555555555666666666677777777778
POSITIONS → 12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-35. Cursor movement (3)

With no interaction with the program, an entire screen of information can be prepared for input, and transferred at one time. This is what is meant by static screen operation, in contrast to the transactional prompt/reply dialogue typical of roll screen operation.

Figure 11-36 shows a completed input screen. The operator is now at the point where the program must be signalled to proceed.

```

LINES
↓
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3
4 CLASS NAME: SERIES/1 HARDWARE  INSTRUCTOR NAME: JOHN JONES
5
6 NAME: JOHN JAMES                STREET: 111 GRANT AVENUE
7                                CITY : ENDICOTT
8                                STATE : NEW YORK 13760
9
10
11 NAME: JAMES JONES              STREET: 255 ALHAMBRA CIRCLE
12                                CITY : CORAL GABLES
13                                STATE : FLORIDA 33135
14
15
16 NAME: JIM JOHNS                STREET: 140 EAST TOWN STREET
17                                CITY : COLUMBUS
18                                STATE : OHIO 43215
19
20
21 NAME: JOAN JIMSON              STREET: 6216 WASHINGTON AVENUE
22                                CITY : RACINE
23                                STATE : WISCONSIN 53406 _
CHARACTER      11111111112222222222333333333344444444445555555555666666666677777777778
POSITIONS → 12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-36. Full screen

In Figure 11-37, the WAIT KEY at WAITONE will be terminated by pressing the ENTER key or a PF key. The computed GOTO following the WAIT KEY will transfer control to various entry points, depending on the return code in "taskname+2." A return code of zero, from the ENTER key, will cause a transfer to location READ. PF1 through PF4 will return codes of 1 through 4, and result in transfers to E1 through E4, respectively. (With the GOTO coded as shown, a PF key higher than PF4 will cause a transfer to READ, as the return code would be outside the valid range of index values 1-4, just as the zero returned by the ENTER key is outside that range, and also results in a transfer to READ.)

For now, assume the operator presses the ENTER key, signalling the program that the page is complete, and transferring control to READ.

```

XMPLSTAT PROGRAM  START
IOCB1     IOCB     NHIST=0
IOCB2     IOCB     SCREEN=
ATTNLIST  (ENTER=4, SPACES=11
          TERMCTRL DISPLAY
WAITONE   WAIT     KEY
          GOTO     (READ,E1,E2,E3,E4),XMPLSTAT+2
          .
          .
          .
READ      QUESTION 'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
          ERASE    MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
          ERASE    MODE=SCREEN,LINE=6
          PRINTX   LINE=6,SPACES=5
          TERMCTRL DISPLAY
          GOTO     WAITONE
CLEANUP   ERASE    MODE=SCREEN,TYPE=ALL
          DEQT
          GOTO    START
          .
          .
          .
          ENDPROG
          END

```

Figure 11-37. ENTER key

In a real program, the routine at location READ would contain the READTEXT instructions necessary to read all the data entered on the screen. In the application illustrated here, that data would presumably be collected and used to print a class roster for the SERIES/1 HARDWARE course taught by JOHN JONES.

Assuming that the contents of the screen has been transferred, the QUESTION instruction at READ displays the prompt message MORE ENTRIES? in the operator guide area at the upper right of the screen, as shown in Figure 11-38.

```

LINES
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1      PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4      MORE ENTRIES ? _
3
4 -----
4 CLASS NAME: SERIES/1 HARDWARE  INSTRUCTOR NAME: JOHN JONES
5 -----
6 NAME:JOHN JAMES                STREET: 111 GRANT AVENUE
7                                CITY  : ENDICOTT
8                                STATE : NEW YORK 13760
9
10
11 NAME: JAMES JONES              STREET: 255 ALHAMBRA CIRCLE
12                                CITY  : CORAL GABLES
13                                STATE : FLORIDA 33135
14
15
16 NAME: JIM JOHNS                STREET: 140 EAST TOWN STREET
17                                CITY  : COLUMBUS
18                                STATE : OHIO 43215
19
20
21 NAME: JOAN JIMSON              STREET: 6216 WASHINGTON AVENUE
22                                CITY  : RACINE
23                                STATE : WISCONSIN 53406
CHARACTER      111111111122222222223333333333444444444455555555556666666667777777778
POSITIONS→12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-38. After ENTER key

The MORE ENTRIES? query is asking the operator, "Are there more students to add to this roster, or are the students just read from the current screen the last ones at this time?"

The QUESTION statement is coded with NO=CLEANUP. YES= is not coded, and therefore a YES response will result in execution of the ERASE instruction following the QUESTION. Assume there are more students, and YES is the response. The first ERASE following the QUESTION clears the prompt and reply from the operator guide area, and the second ERASE clears all unprotected data from the four data entry areas in lines 6 through 23. The SERIES/1 HARDWARE and JOHN JONES entries in the header area are left undisturbed, since the student names and addresses to be entered are still for the same class.

The PRINTEXT following the second ERASE (Figure 11-37) positions the cursor at the first unprotected entry field for the first data entry area. The TERMCTRL DISPLAY that follows displays the cursor, resulting in the screen shown in Figure 11-39.

```

LINES
↓
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3 -----
4 CLASS NAME SERIES/1 HARDWARE   INSTRUCTOR NAME JOHN JONES
5 -----
6 NAME: _                         STREET:
7                                CITY :
8                                STATE :
9
10
11 NAME:                            STREET:
12                                CITY :
13                                STATE :
14
15
16 NAME:                            STREET:
17                                CITY :
18                                STATE :
19
20
21 NAME:                            STREET:
22                                CITY :
23                                STATE :
CHARACTER      111111111122222222223333333333334444444444555555555566666666667777777778
POSITIONS → 12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-39. Reply YES to QUESTION

If there were no more students to enter for this roster, and the response to the MORE ENTRIES? prompt were NO, the QUESTION statement (Figure 11-37) would transfer control to location CLEANUP, which erases both protected and unprotected areas of the entire screen, dequeues the terminal, and goes back to the beginning of the program (START), bringing up the roll screen with the initial operator directions, as shown in Figure 11-40.

```

LINES
↓
0
1 CLASS ROSTER PROGRAM
2
3 HIT 'ATTN' AND ENTER 'END' TO END THE PROGRAM
4
5 HIT ANY PROGRAM FUNCTION KEY TO BRING UP THE ENTRY SCREEN
6
7 _
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
CHARACTER      111111111122222222223333333333334444444444555555555566666666667777777778
POSITIONS → 12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-40. Reply NO to QUESTION

In Figure 11-41, assume the program is again suspended by the WAIT KEY at WAITONE, with the completed screen depicted in Figure 11-36. The transfer to location READ and the MORE ENTRIES? prompt from the QUESTION statement resulted from the operator's pressing the ENTER key. The WAIT KEY may also be terminated by a PF key.

There are no pre-assigned functions for PF keys, other than the hardcopy facility already discussed. Therefore, the purpose of a particular PF key in any program is defined by the instructions coded in the routine to which control is transferred when that PF key is depressed.

In the example in Figure 11-41, PF1 through PF4 have been assigned as delete functions for the four data entry areas, as shown by the operator guides at the top of the screen (Figure 11-36).

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIO
ATTNLIST (END OF
=4, SPACES=11
TERMCTRL DISPLAY
WAITONE WAIT KEY
GOTO (READ,E1,E2,E3,E4),XMPLSTAT+2
E1 MOVE LINENBR,6
GOTO DELETE
E2 MOVE LINENBR,11
GOTO DELETE
E3 MOVE LINENBR,16
GOTO DELETE
E4 MOVE LINENBR,21
DELETE ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
ADD LINENBR,1
ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
ADD LINENBR,1
ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
SUBTRACT LINENBR,2
PRINTTEXT LINE=LINENBR,SPACES=5
TERMCTRL DISPLAY
GOTO WAITONE
.
.
.
LINENBR DATA F'0'
ENDPROG
END

```

Figure 11-41. PF keys

Assume that for some reason, the student JIM JOHNS, the third entry on the screen, is not supposed to be on the class roster; the operator, therefore, presses PF3.

In Figure 11-41, the PF key terminates the WAIT KEY, and the computed GOTO transfers control to E3. The MOVE at E3 initializes the LINENBR variable to 16, which is the top line of the third data entry area. Control is then transferred to DELETE, where successive ERASE operations and adjustments of the LINENBR variable result in erasure of the unprotected portions of the third data entry area. Before returning to the WAIT KEY, the cursor is positioned and displayed at the first entry field of the erased data area, as shown in Figure 11-42.

```

LINES
0
1 ENTER KEY = PAGE COMPLETE      PF1 = DELETE ENTRY 1    PF2 = DELETE ENTRY 2
2 PF3 = DELETE ENTRY 3          PF4 = DELETE ENTRY 4
3 -----
4 CLASS NAME: SERIES/1 HARDWARE  INSTRUCTOR NAME: JOHN JONES
5 -----
6 NAME: JOHN JAMES              STREET: 111 GRANT AVENUE
7                                CITY : ENDICOTT
8                                STATE : NEW YORK 13760
9
10
11 NAME: JAMES JONES            STREET: 255 ALHAMBRA CIRCLE
12                                CITY : CORAL GABLES
13                                STATE : FLORIDA 33135
14
15
16 NAME: _                       STREET:
17                                CITY :
18                                STATE :
19
20
21 NAME: JOAN JIMSON            STREET: 6216 WASHINGTON AVENUE
22                                CITY : RACINE
23                                STATE : WISCONSIN 53406
CHARACTER      11111111112222222222333333333344444444445555555555666666666677777777778
POSITIONS→12345678901234567890123456789012345678901234567890123456789012345678901234567890

```

Figure 11-42. After PF3

For your reference, the program example used in the foregoing discussion is shown in its entirety in Figure 11-43.

```

XMPLSTAT PROGRAM START
IOCB1 IOCB NHIST=0
IOCB2 IOCB SCREEN=STATIC
ATTNLIST (END,OUT,$PF,STATIC)
START ENQT IOCB1
PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
PRINTTEXT ' THE PROGRAM'
PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
PRINTTEXT ' BRING UP THE ENTRY SCREEN'
DEQT
CHECK WAIT ATTNECB,RESET
IF (ATTNECB,EQ,1),GOTO,ENDIT
ENTRY ENQT IOCB2
ERASE MODE=SCREEN,TYPE=ALL
TERMCTRL BLANK
PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
PRINTTEXT ' PF1 = DELETE ENTRY 1'
PRINTTEXT ' PF2 = DELETE ENTRY 2'
PRINTTEXT 'PF3 = DELETE ENTRY 3 ',SKIP=1
PRINTTEXT 'PF4 = DELETE ENTRY 4'
HDR PRINTTEXT DASHES,PROTECT=YES,LINE=3
PRINTTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
PRINTTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
PRINTTEXT DASHES,PROTECT=YES,LINE=5
MOVE LINENBR,6
DO 4,TIMES
PRINTTEXT 'NAME:',LINE=LINENBR,PROTECT=YES
PRINTTEXT 'STREET:',LINE=LINENBR,SPACES=30,PROTECT=YES
A1 ADD LINENBR,1
PRINTTEXT 'CITY :',LINE=LINENBR,SPACES=30,PROTECT=YES
A2 ADD LINENBR,1
PRINTTEXT 'STATE :',LINE=LINENBR,SPACES=30,PROTECT=YES
ADD LINENBR,3
ENDDO
PRINTTEXT LINE=4,SPACES=11
TERMCTRL DISPLAY
WAITONE WAIT KEY
GOTO (READ,E1,E2,E3,E4),XMPLSTAT+2

```

Figure 11-43. Complete program (1 of 2)

```

E1      MOVE      LINENBR,6
        GOTO      DELETE
E2      MOVE      LINENBR,11
        GOTO      DELETE
E3      MOVE      LINENBR,16
        GOTO      DELETE
E4      MOVE      LINENBR,21
DELETE  ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
        ADD       LINENBR,1
        ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
        ADD       LINENBR,1
        ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
        SUBTRACT  LINENBR,2
        PRINTTEXT LINE=LINENBR,SPACES=5
        TERMCTRL  DISPLAY
        GOTO      WAITONE
READ    QUESTION  'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
        ERASE     MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
        ERASE     MODE=SCREEN,LINE=6
        PRINTTEXT LINE=6,SPACES=5
        TERMCTRL  DISPLAY
        GOTO      WAITONE
CLEANUP ERASE     MODE=SCREEN,TYPE=ALL
        DEQT
        GOTO      START
ENDIT   PROGSTOP
        DATA     X'5050'
DASHES  DATA     80C'-'
OUT     POST      ATTNECB,1
        ENDATTN
STATIC  POST      ATTNECB,-1
        ENDATTN
ATTNECB ECB
LINENBR DATA     F'0'
        ENDPROG
        END

```

Figure 11-43. Complete program (2 of 2)

## RDCURSOR INSTRUCTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "RDCURSOR."

Another instruction applying only to static screens, but not used in the foregoing programming example, is RDCURSOR. This instruction will store the line number and indent from the left margin (SPACES) corresponding to the current cursor position, in user program variables. It can be used as an additional means of communication between program and operator. For example, if a prompt displayed on a particular screen is unusually cryptic, an operator unfamiliar with the application might not know what data should be entered in the associated data entry field. If a particular PF key is designated as the help function, and results in a transfer to a routine which executes a RDCURSOR instruction, the operator can position the cursor in the data entry field whose purpose is in doubt, and press the help PF key. The RDCURSOR command could then sense the cursor position, find out which field is causing the confusion by comparing the sensed position to the known data entry field locations, and display explicit instructions for the field in question.

## PRINTNUM/GETVALUE INSTRUCTIONS

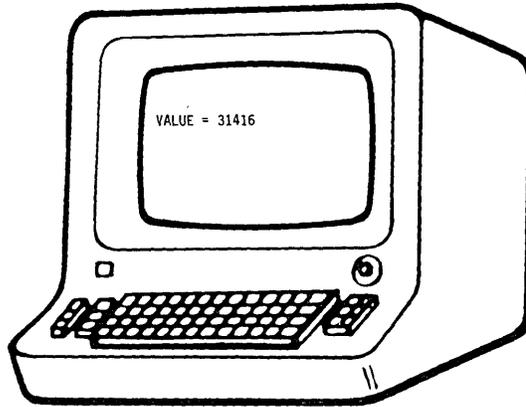
READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "PRINTNUM", "GETVALUE."

The PRINTEXT and READTEXT instructions are used to transfer EBCDIC character strings to and from terminals. PRINTNUM and GETVALUE instructions perform the same functions for numeric values. PRINTNUM takes a numeric value in storage, automatically performs the conversion from internal (binary) representation, and transfers it to a terminal for display or printing.

PRINTNUM can display a single value,

```
PRINTNUM loc
```

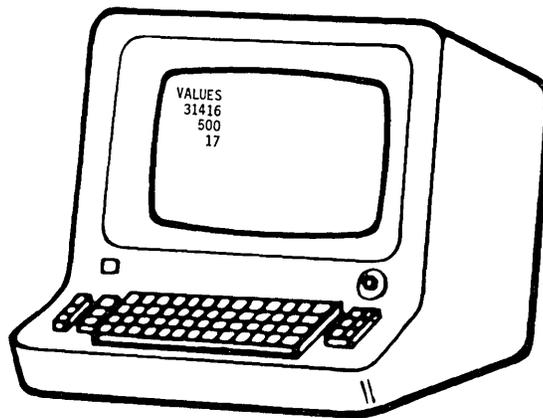
```
P1      PROGRAM  START
START  PRINTTEXT 'VALUE = '
        PRINTNUM IVAL
        PRINTTEXT SKIP=1
        PROGSTOP
IVAL    DATA    F'31416'
        ENDPROG
        END
```



– or a single PRINTNUM statement can be used to display multiple values. When more than one value is displayed by the same PRINTNUM, the values can be displayed on separate lines,

```
PRINTNUM loc,count,nline
```

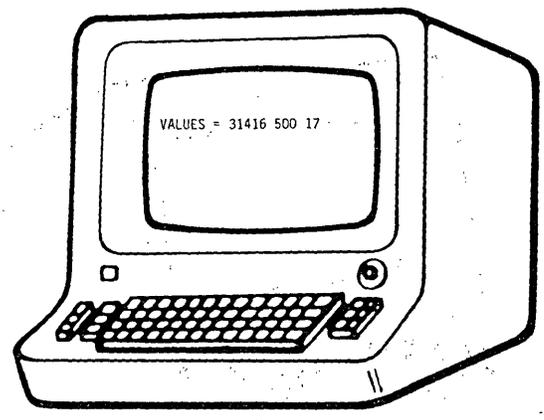
```
P1      PROGRAM  START
START  PRINTEXT  'VALUES'
        PRINTNUM IVALS,3,1,SKIP=1
        PRINTEXT SKIP=1
        PROGSTOP
IVALS  DATA    F'31416'
        DATA    F'500'
        DATA    F'17'
        ENDPROG
        END
```



— or can be displayed on the same line.

```
PRINTNUM loc,count,nline
```

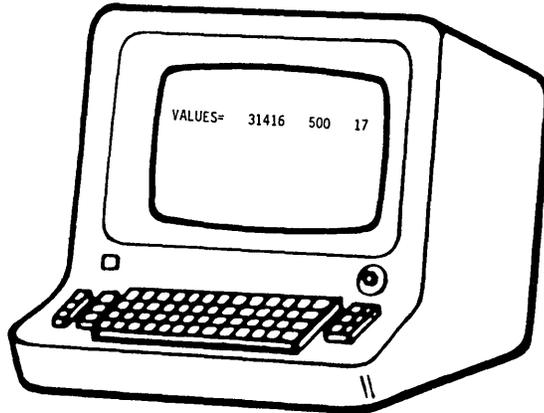
```
P1      PROGRAM      START
START  PRINTTEXT    'VALUES'
        PRINTNUM    IVALS,3,3,SKIP=1
        PRINTTEXT    SKIP=1
        PROGSTOP
IVALS   DATA       F'31416'
        DATA       F'500'
        DATA       F'17'
        ENDPROG
        END
```



When multiple values appear on the same line, you can control the spacing between values.

```
PRINTNUM loc,count,nline,nspace
```

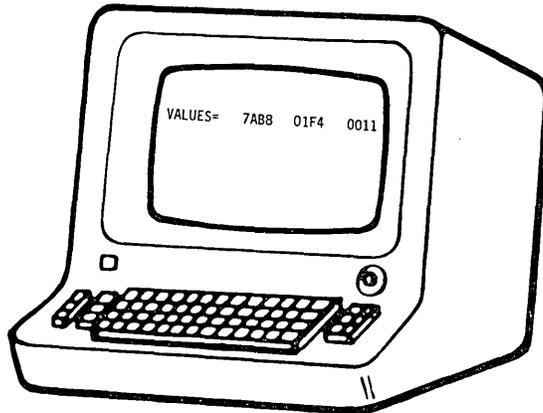
```
P1 PROGRAM START
START PRINTTEXT 'VALUES = '
PRINTNUM IVALS,3,3,10
PRINTTEXT SKIP=1
PROGSTOP
IVALS DATA F'31416'
DATA F'500'
DATA F'17'
ENDPROG
END
```



If desired, values may be displayed in hexadecimal rather than decimal form.

```
PRINTNUM loc,count,nline,space,MODE=
```

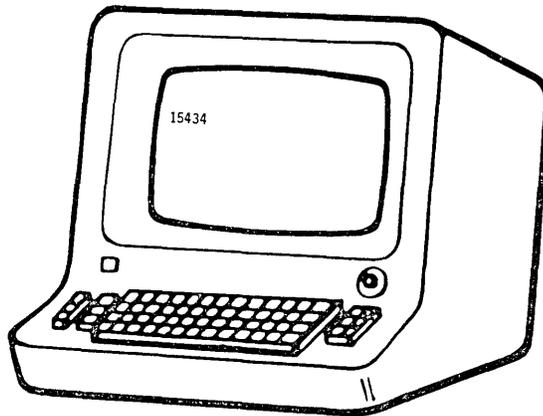
```
P1 PROGRAM START
START PRINTTEXT 'VALUES = '
PRINTNUM IVALS,3,3,10,MODE=HEX
PRINTTEXT SKIP=1
PROGSTOP
IVALS DATA F'31416'
DATA F'500'
DATA F'17'
ENDPROG
END
```



GETVALUE transfers a numeric text string, input by an operator, into storage, automatically converting to internal (binary) representation.

GETVALUE 10c

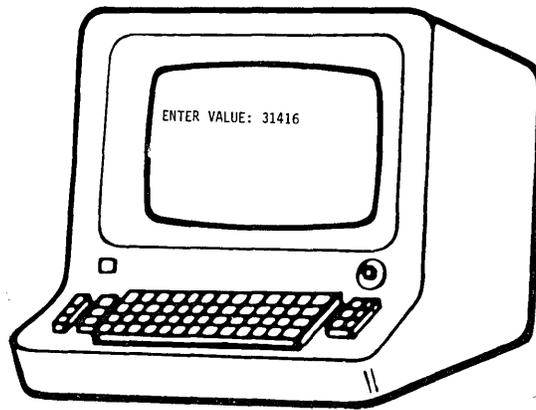
P1	PROGRAM	START
START	GETVALUE	IVAL
	PROGSTOP	
IVAL	DATA	F'0'
	ENDPROG	
	END	



As with READTEXT, a prompt message may be issued prior to the input operation.

```
GETVALUE loc,pmsg
```

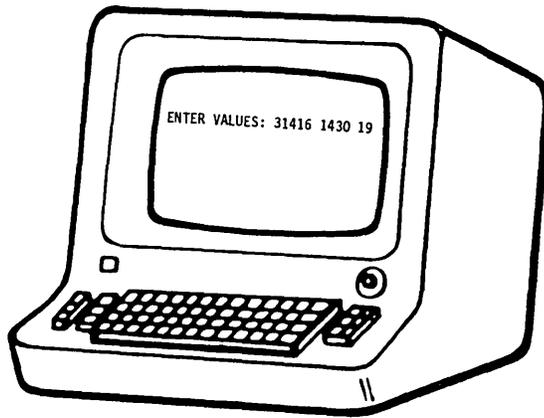
```
P1      PROGRAM  START
START  GETVALUE IVAL,'ENTER VALUE:'
        PROGSTOP
IVAL   DATA    F'0'
        ENDPROG
        END
```



Multiple values can be read by a single GETVALUE instruction,

```
GETVALUE loc,pmsg,count
```

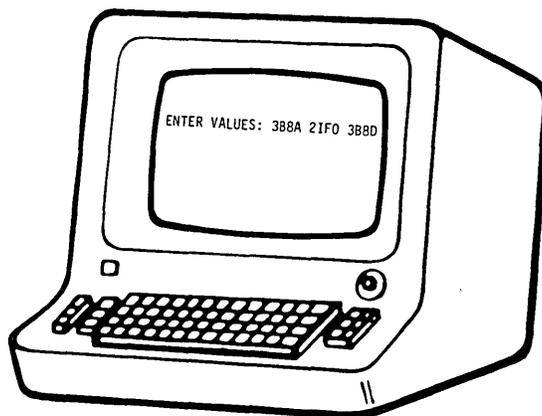
```
P1    PROGRAM  START
START GETVALUE IVALS,'ENTER VALUES:',3
      PROGSTOP
IVAL  DATA    3F'0'
      ENDPROG
      END
```



– and hexadecimal input can be accepted.

```
GETVALUE 1oc,pmsg,count,MODE=
```

```
P1      PROGRAM  START
START  GETVALUE IVALS,'ENTER VALUES:',3,MODE=HEX
        PROGSTOP
IVALS  DATA    3F'0'
        ENDPROG
        END
```



Forms control operands (SKIP=, LINE=, and SPACES=) serve the same purpose and are used the same way with PRINTNUM and GETVALUE as for PRINTTEXT and READTEXT. See the reading assignment for how to use PRINTNUM and GETVALUE with double precision integers, standard and extended precision floating point values, and the external data formatting option.

## PRINTIME/PRINDATE INSTRUCTIONS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "PRINTIME", "PRINDATE."

PRINTIME and PRINDATE are pre-defined terminal output operations. PRINTIME will display the current value of the system 24 hour clock in the format HH:MM:SS. PRINDATE displays the date as MM/DD/YY or DD/MM/YY depending on the option selected on the SYSTEM statement when the supervisor was generated.

## TERMINAL I/O REVIEW EXERCISE – QUESTIONS

1. Describe the program states or conditions which, while in effect, inhibit the ATTNLIST capability.

a. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

b. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

2. List three buffer forcing conditions.

a. \_\_\_\_\_

b. \_\_\_\_\_

c. \_\_\_\_\_

3. Assume the following two instructions are executed, directed at a static screen.

```
PRINTTEXT 'ENTER:      ',LINE=3,PROTECT=YES  
PRINTTEXT 'NEXT ENTRY: ',SPACES=10,PROTECT=YES
```

What character position will the N in NEXT occupy?

Answer: \_\_\_\_\_

4. On the left are listed the interrupt generating terminal keys. In the space following each key, list the letter(s) designating the statement(s) on the right that apply to each key. More than one statement may be true for each key, and each statement may apply to more than one key.

PF keys_____	a. will terminate a WAIT KEY operation
ATTN key_____	b. used with ATTNLIST, not with WAIT KEY
ENTER key_____	c. used with WAIT KEY, never with ATTNLIST
	d. will <i>not</i> terminate a WAIT KEY operation
	e. can be used with ATTNLIST, and will also terminate a WAIT KEY

5. List the special system terminals that may be enqueued by coding their names as the operand of an ENQT instruction.

Answer: \_\_\_\_\_

**This page intentionally left blank.**

6. Below on the left is a list of five operator entries. Each entry is in response to the GETVALUE prompt in the program given.

On the right are spaces for the values that would be displayed by execution of the PRINTNUM immediately following the GETVALUE in the program. Fill in what the PRINTNUM would display after each of the entries on the left (each operator entry/PRINTNUM display pair should be considered a new load/execution of the program).

```

P1      PROGRAM  START
START   GETVALUE VAL, 'ENTER NBR: '
        PRINTNUM VAL
        PRINTTEXT SKIP=1
        PROGSTOP
VAL     DATA    F'0'
        ENDPROG
        END

```

OPERATOR ENTRY	PRINTNUM DISPLAY
a. 1492	_____
b. -3	_____
c. 39000	_____
d. NO ENTRY (ENTER KEY ONLY)	_____
e. 1BA3	_____

## TERMINAL I/O REVIEW EXERCISE – ANSWERS

1.
  - a. program has the terminal enqueued
  - b. program is suspended by a WAIT KEY operation
2. Any three of the following:
  - a. "LINE=" in a succeeding operation
  - b. "SKIP=" in a succeeding operation
  - c. DEQT execution
  - d. an "@" character imbedded in the text of this or of a succeeding operation, with MODE=WORD in effect
  - e. TERMCTRL DISPLAY execution
  - f. "change of operation direction", such as a PRINTTEXT followed by a GETVALUE or READTEXT
3. Character position 21, line 3. The "SPACES=10" leaves 10 unprotected spaces between the end of the preceding protected field, and the beginning of the "NEXT ENTRY" text.
4. PF keys a, e PF keys (a) will terminate a WAIT KEY operation, and, when a program is not suspended by a WAIT KEY, and the terminal is not enqueued, may also be used in an ATTNLIST (e).

ATTN key b, d The ATTN key will not terminate a WAIT KEY operation (d). When the program is not in a WAIT KEY, and the terminal is not enqueued, the ATTN key may be used by the ATTNLIST function (b).

ENTER key a, c The ENTER key terminates a WAIT KEY (a) (as well as the implied wait of a READTEXT/GETVALUE/QUESTION), and cannot be used with ATTNLIST (c).
5. Answer: \$\$SYSPRTR, \$\$SYSLOG The third "special system terminal", \$\$SYSLOGA may be enqueued by user programs, but only by using the "ENQT/label of IOCB" convention, or by an ENQT with no IOCB label reference, when \$\$SYSLOGA is the "loading" terminal.

6. OPERATOR ENTRY	PRINTNUM DISPLAY
a. 1492	<u>1492</u>
b. -3	<u>-3</u>
c. 39000	<u>0</u>
d. NO ENTRY (ENTER KEY ONLY)	<u>0</u>
e. 1BA3	<u>1</u>

Entries a. and b. operate normally. Entry c. is too large to be contained in a single word integer, so VAL is left undisturbed, as it is for d., when no entry is made. Entry e. is an attempt to enter a hexadecimal value, when "MODE=HEX" is not coded in the GETVALUE operand field. The input operation terminates when the first non-numeric character is encountered in the input field.

This page intentionally left blank.

## Section 12. Data Formatting

OBJECTIVES: After completing this topic, the student should

1. Understand when to use the data formatting/conversion instructions
2. Be able to convert numeric character strings to binary values using CONVTD
3. Be able to convert binary values to EBCDIC character strings using CONVTB
4. Understand the operation of GETEDIT/PUTEDIT instructions, and their relationship to FORMAT and TEXT statements

### DATA CONVERSION

For purposes of this discussion, data conversion refers to the process of converting arithmetic values from internal representation (binary) into external representation (EBCDIC character strings), or the reverse.

You are already familiar with some forms of data conversion. As illustrated in Figure 12-1, the assembler performs data conversion when assembling arithmetic constants, defined in DATA statements. The binary values generated during the assembly are the internal equivalents of the externally represented values coded in the source statements.

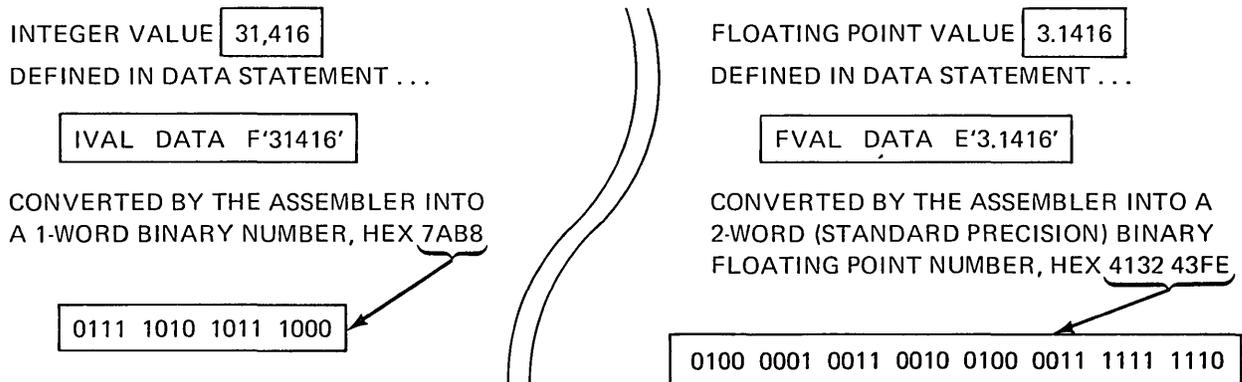


Figure 12-1. Assembler data conversion



The PREC= operand describes opnd1 (Figure 12-3).

PREC= Operand	opnd1 Description	Storage Required
PREC=S	Single Precision Integer (default)	1 Word (2 Bytes)
PREC=D	Double Precision Integer	2 Words (4 Bytes)
PREC=F	Standard Precision Floating Point	2 Words (4 Bytes)
PREC=L	Extended Precision Floating Point	4 Words (8 Bytes)

Figure 12-3. PREC= operand

The FORMAT= operand is coded as a list containing three sublist elements, all enclosed in parentheses. The three elements describe the EBCDIC character string pointed to by the label in opnd2, as shown in Figure 12-4.

FORMAT=(W,D,T) where;

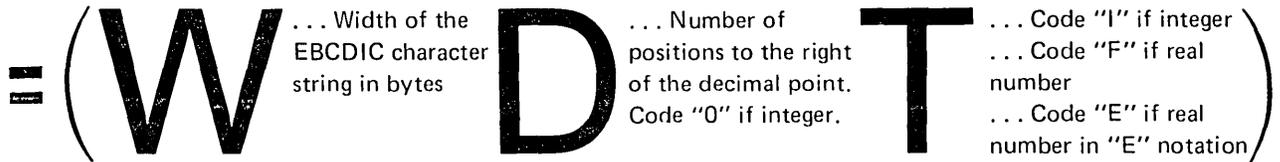


Figure 12-4. FORMAT= operand

If not coded, FORMAT= defaults to FORMAT=(6,0,I), indicating a six-byte EBCDIC field containing an integer number.

## CONVTB INSTRUCTION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "CONVTB."

CONVTB converts values in internal representation (binary) form to an EBCDIC character string.

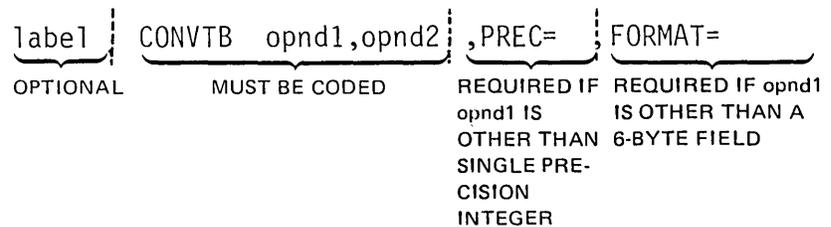


Figure 12-5. CONVTB format

Since the direction of the operation is the reverse of CONVTD, the meaning of opnd1 and opnd2 is also reversed. The label of the left-most byte of the storage area, which will receive the EBCDIC string resulting from the conversion, is opnd1 and opnd2 is the label of the storage location containing the variable.

The PREC= and FORMAT= operands are coded the same way for CONVTB as for CONVTD; because opnd1 and opnd2 are reversed, PREC= now applies to opnd2 and FORMAT= to opnd1.

## CONVTD/CONVTB CODING EXAMPLES

In Figure 12-6, the CONVTB at C1 is converting the constant at location CON1 into an EBCDIC character string, which will be stored in the text buffer EBC1.

```

CCODE    PROGRAM    C1
C1       CONVTB     EBC1,CON1
          IF         (CCODE,NE,-1),GOTO,CNVTERR
P1       PRINTTEXT  'TEXT='
          PRINTTEXT  EBC1
          PRINTTEXT  SKIP=1
END      PROGSTOP
CNVTERR  MOVE       CODE,CCODE
          PRINTTEXT  ' CONVERT ERROR, CODE='
          PRINTNUM   CODE
          PRINTTEXT  SKIP=1
          GOTO       END
EBC1    TEXT        LENGTH=6
CON1    DATA       F'14398'
CODE    DATA       F'0'
          ENDPROG
          END

```

Figure 12-6. Return code = -1

Completion codes for CONVTB and CONVTD operations are returned in the task code word (taskname). The IF statement immediately following the CONVTB is checking the return code for Normal Completion (-1). In this example, the operation will be successful, and the PRINTTEXT instructions beginning at P1 will display TEXT=14398.

In Figure 12-7, the CONVTB is attempting to convert a value of 21,000,000, in location CON2, and store the resulting text string in the text buffer at EBC2. The text buffer is not large enough to hold the character string generated by the conversion, and will be set to zeros. The completion code will be a 3, indicating Conversion Error, and the IF statement following the CONVTB will transfer control to location CNVTERR.

The error routine beginning at CNVTERR will display an error message and the completion code resulting from the operation. The first instruction moves the completion code from taskname into the user-defined program variable CODE.

```

CCODE    PROGRAM    C2
          CONVTB    EBC2,CON2,PREC=DWORD
          IF        (CCODE,NE,-1),GOTO,CNVTERR
P1        PRINTTEXT 'TEXT='
          PRINTTEXT EBC2
          PRINTTEXT SKIP=1
END       PROGSTOP
CNVTERR  MOVE        CODE,CCODE
          PRINTTEXT 'CONVERT ERROR, CODE='
          PRINTNUM  CODE
          PRINTTEXT SKIP=1
          GOTO      END
EBC2     TEXT        LENGTH=6
CON2     DATA       D'21000000'
CODE     DATA       F'0'
          ENDPROG
          END

```

Figure 12-7. Return code = 3.

This is a standard convention, and is necessary because other operations, such as I/O, also post completion codes in taskname, and will overlay the code you want to display. For instance, were the IF statement following the CONVTB replaced by the statement

```

PRINTNUM  CCODE

```

in an attempt to display the return code from the conversion operation, the code displayed would be the completion code resulting from execution of the PRINTNUM itself, not the code returned by the CONVTB.

When the error routine at CNVTERR completes execution, the message CONVERT ERROR, CODE=3 will be displayed. A -1, for Normal Completion, or a 3, indicating Conversion Error, are the only completion codes generated by CONVTD operations.

In Figure 12-8, a CONVTD operation is attempting to convert the EBCDIC string in EBC3 to a binary value to be stored in location CON3. The EBCDIC string consists of blanks and the delimiter “,”. This results in no conversion, and a completion code of 2, indicating Field Omitted. Commas and slashes (/) are considered arithmetic delimiters and, if found in a text string during CONVTD execution, will terminate the conversion. In this example, since the delimiter (comma) was preceded only by blanks, the Field Omitted completion code is generated and the program will complete execution with CONVERT ERROR, CODE=2 displayed.

```

CCODE    PROGRAM    C3
C3       CONVTD     CON3,EBC3
          IF        (CCODE,NE,-1),GOTO,CNVTERR
P1       PRINTTEXT  'VARIABLE='
          PRINTNUM  CON3
          PRINTTEXT  SKIP=1
END      PROGSTOP
CNVTERR  MOVE       CODE,CCODE
          PRINTTEXT  'CONVERT ERROR, CODE='
          PRINTNUM  CODE
          PRINTTEXT  SKIP=1
          GOTO      END
EBC3    TEXT        ' , ' , LENGTH=6
CON3    DATA       F'0'
CODE    DATA       F'0'
          ENDPROG
          END

```

Figure 12-8. Return code = 2

If the text buffer at EBC3 had contained numbers (in EBCDIC code), all numbers to the left of the delimiter would have been converted, and a completion code of -1 returned. For instance, 12,391 in the text buffer would convert to the binary equivalent of 12. Any non-numeric character imbedded within the text field will end the conversion.

In Figure 12-9, the CONVTD at C4 is attempting to convert the blank text field at EBC4. This will result in a return code of +1, which indicates No Data In Field. The example will complete with the message CONVERT ERROR, CODE=1 displayed.

```

CCODE    PROGRAM    C4
C4       CONVTD     CON4,EBC4
          IF        (CCODE,NE,-1),GOTO,CNVTERR
P1       PRINTTEXT  'VARIABLE='
          PRINTNUM  CON4
          PRINTTEXT  SKIP=1
END      PROGSTOP
CNVTERR  MOVE       CODE,CCODE
          PRINTTEXT  'CONVERT ERROR, CODE='
          PRINTNUM  CODE
          PRINTTEXT  SKIP=1
          GOTO      END
EBC4    TEXT        LENGTH=6
CON4    DATA       F'0'
CODE    DATA       F'0'
          ENDPROG
          END

```

Figure 12-9. Return code = 1

## GETEDIT/PUTEDIT INTRODUCTION

GETEDIT and PUTEDIT instructions combine several of the I/O and conversion operations already discussed. For review, Figure 12-10 summarizes the instructions used to move data from a terminal into storage (READTEXT, GETVALUE) and convert it to internal representation (CONVTD, or implicit with GETVALUE).

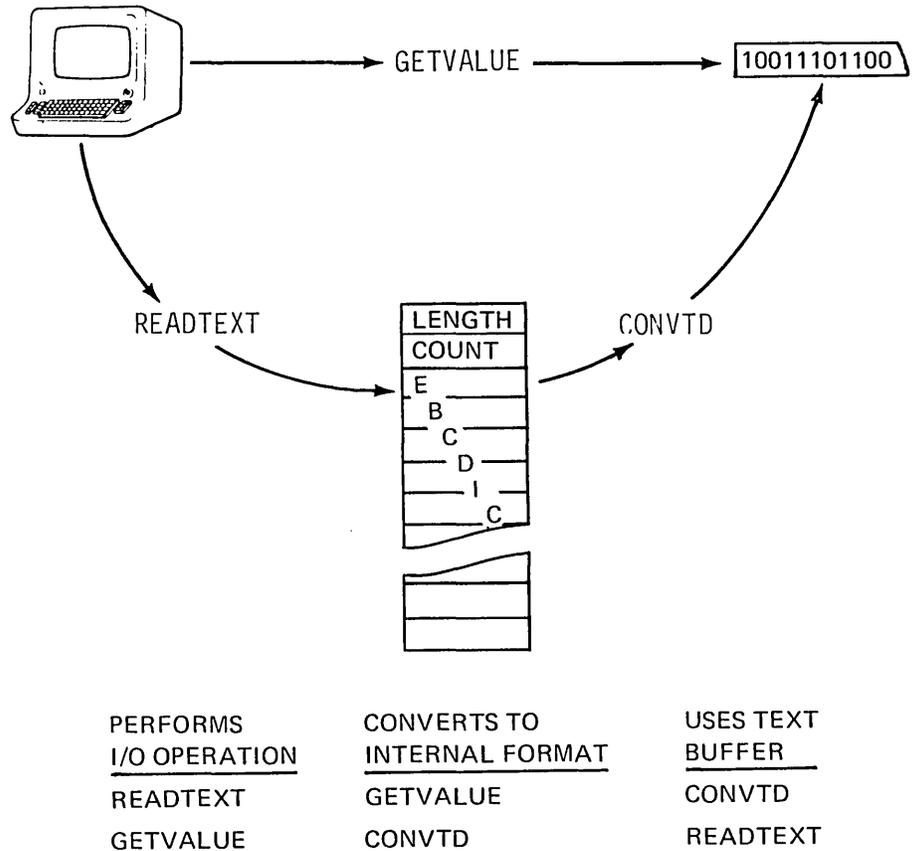
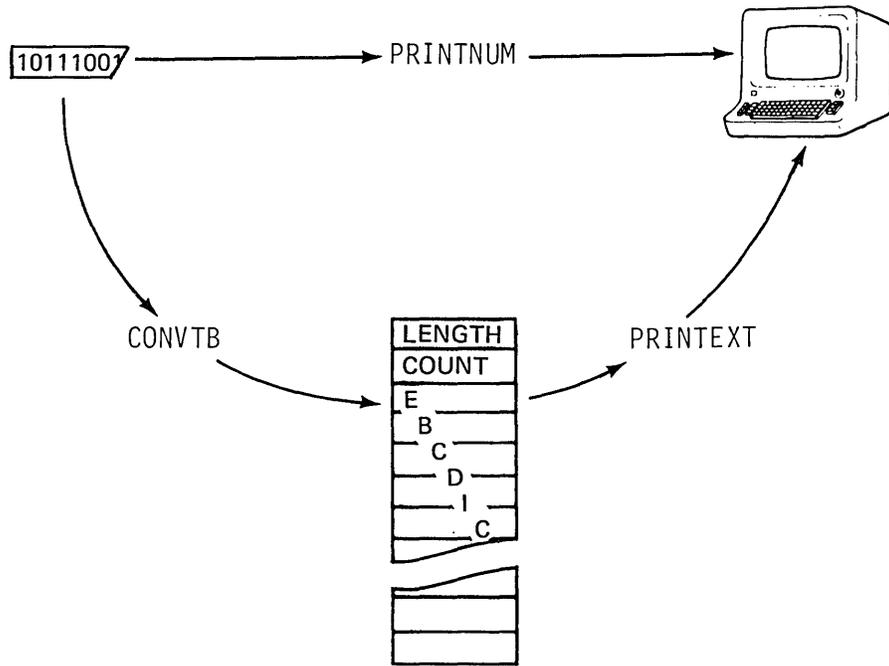


Figure 12-10. External to internal summary

In Figure 12-11, the reverse operations are shown, converting and moving data directly to a terminal (PRINTNUM), or first converting it to external format (CONVTB), and then displaying it (PRINTEXT).



PERFORMS  
I/O OPERATION

PRINTEXT  
PRINTNUM

CONVERTS TO  
EXTERNAL FORMAT

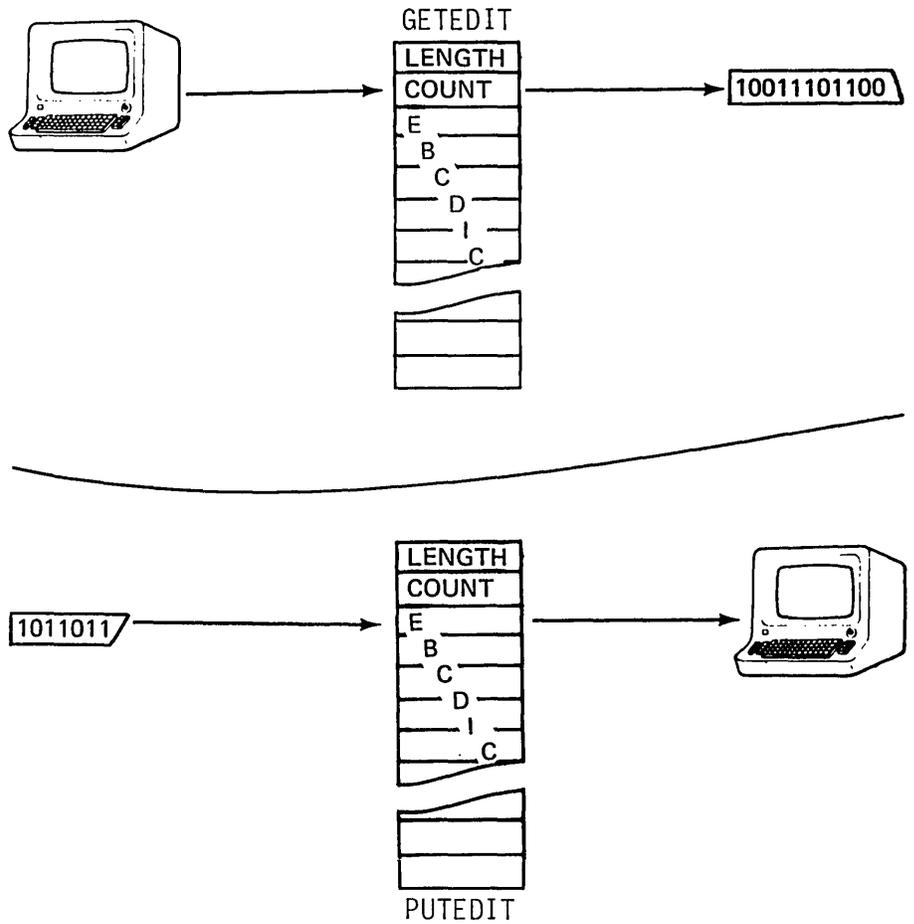
PRINTNUM  
CONVTB

USES TEXT  
BUFFER

CONVTB  
PRINTEXT

Figure 12-11. Internal to external summary

PUTEDIT and GETEDIT perform all of the functions shown in Figures 12-10 and 12-11. The I/O plus conversion provided by GETVALUE and PRINTNUM is supported, but with the addition of the use of a text buffer. The value is therefore displayed/read (I/O), and is available both in external format (as EBCDIC string in text buffer) and in internal format.



1. Performs I/O operation (optional)
2. Performs conversion
3. Uses text buffer

Figure 12-12. PUTEDIT/GETEDIT summary

Viewed another way, the transfer of an EBCDIC string to or from a terminal as provided by PRINTTEXT and READTEXT is supported, but with the addition of conversion to or from internal representation (CONVTD/CONVTB functions).

## PUTEDIT/GETEDIT INSTRUCTIONS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "GETEDIT", "PUTEDIT."

To perform a conversion, four items of information are required:

1. Direction of conversion (from internal representation to external, or the reverse). This is implicit when GETEDIT (external to internal) or PUTEDIT (internal to external) is coded.
2. Conversion specification. Length of character string and type of data item to be converted to or from. This information is coded in a FORMAT statement, and the location (label) of the FORMAT statement is the first operand of the GETEDIT or PUTEDIT.
3. Character buffer location. The second operand is the name of the character buffer (usually the label of a TEXT statement) that contains the character string to be converted (GETEDIT) or will hold the results of the conversion (PUTEDIT).
4. Storage variable location. The named program storage location(s) containing the internally represented data item(s) that are the input to (PUTEDIT) or results of (GETEDIT) the conversion. Figure 12-13 summarizes the operand format just discussed, using GETEDIT as an example. (GETEDIT is used in most of the following illustrations, but the concepts demonstrated are equally valid for PUTEDIT operations, if the direction of conversion is taken into account.)

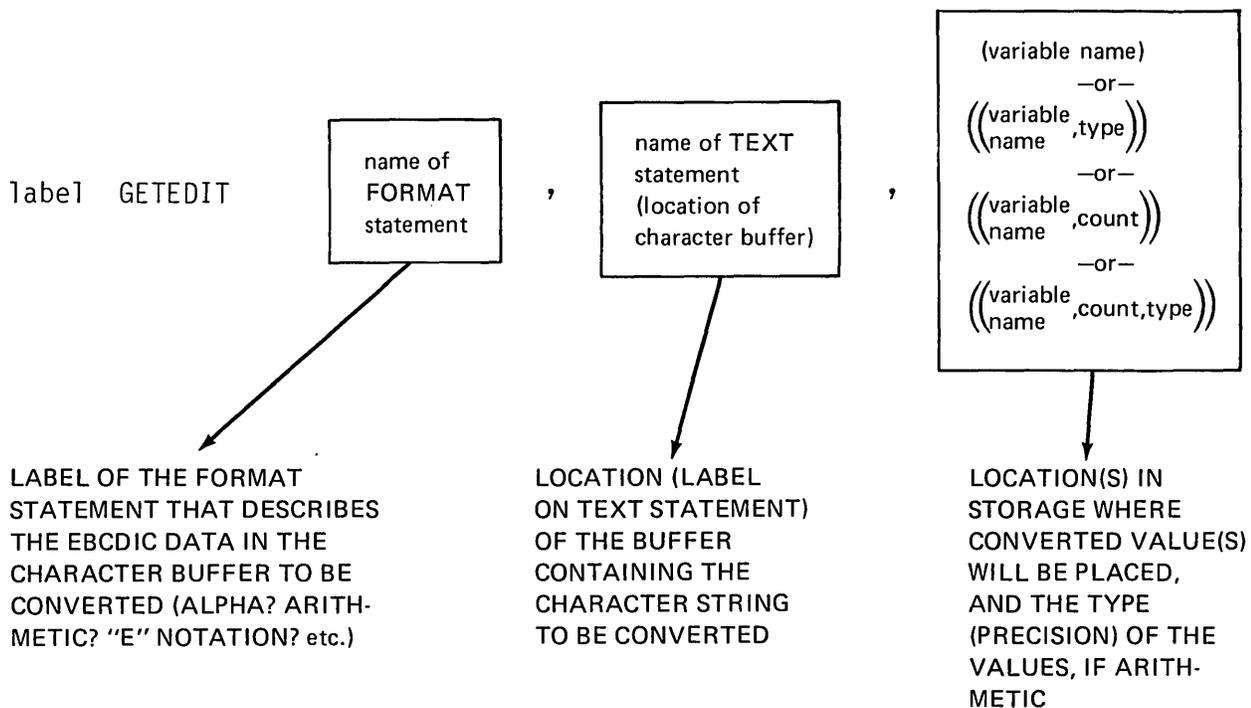


Figure 12-13. GETEDIT format

# FORMAT STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "FORMAT."

Figure 12-14 illustrates the basic layout of the FORMAT statement, and shows how it is referenced by a GETEDIT.

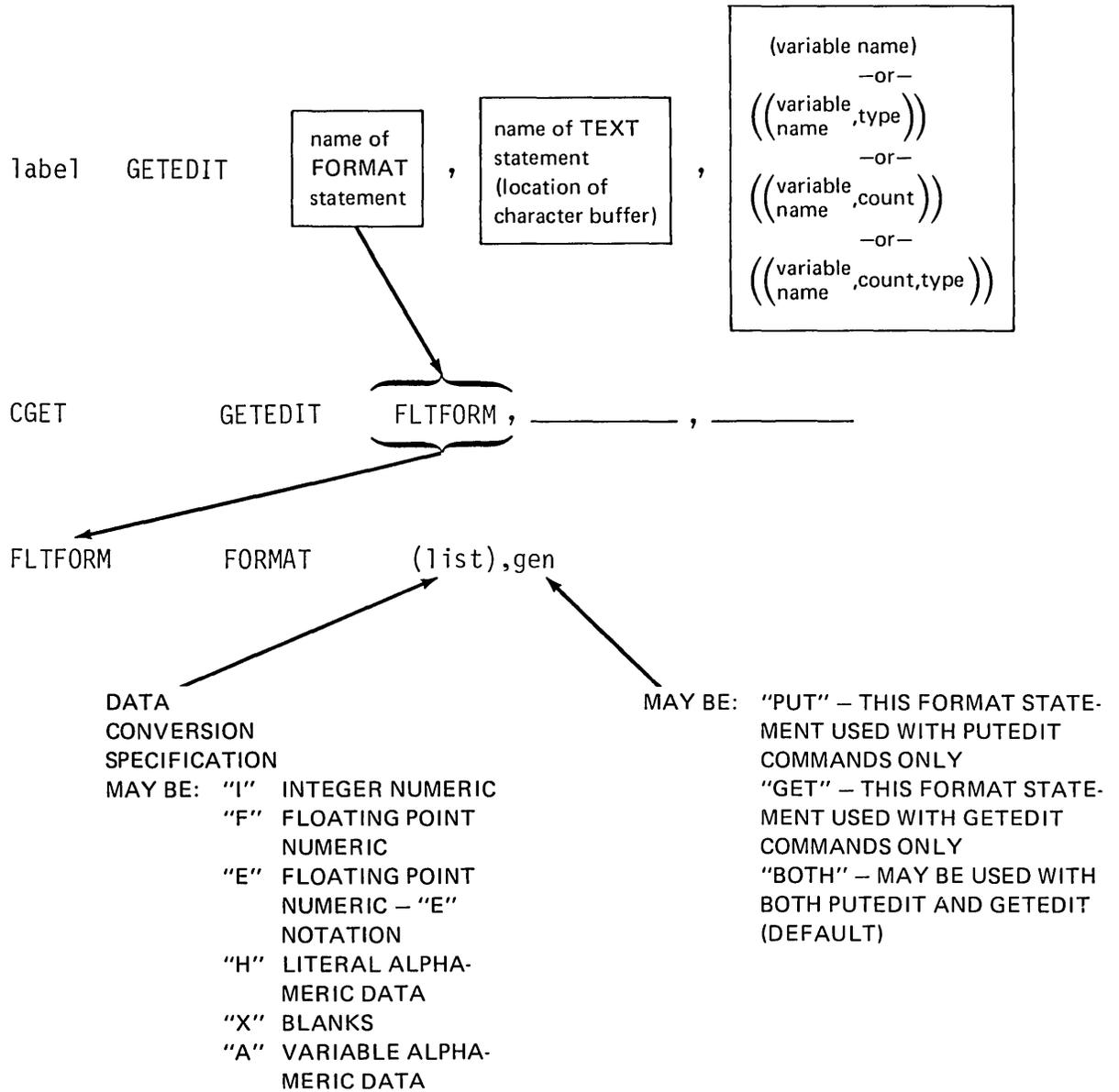


Figure 12-14. FORMAT statement

Note that among the various types of data items that are allowed in the data conversion specification list are type F and type E. The type F indicates *floating point numeric*. Do not confuse this with the *fixed point binary* designated by the F that is used in DATA statements. Similarly, the E means *E-type notation*, and not *standard precision floating point*, as did the E used with DATA statements. By specifying E-type notation in the FORMAT list, the variable being described is implicitly considered to be a floating point value.

Figure 12-15 is an example of a FORMAT statement, whose list describes a single variable, with data conversion specification type E. Detailed explanations of all the available data specification types, and examples of their use, may be found in the reading assignment.

**FORMAT**

- SPECIFIES THE TYPE OF CONVERSION TO BE PERFORMED WHEN DATA IS TRANSFERRED FROM STORAGE TO A TEXT BUFFER BY A PUTEDIT COMMAND, OR FROM A TEXT BUFFER TO STORAGE BY A GETEDIT COMMAND.

**EXAMPLE:** WRITE A FORMAT STATEMENT THAT WILL ALLOW CONVERSION TO AND FROM FLOATING POINT NUMBERS WITHIN THE RANGE OF -9.9999 TO +9.9999, USING "E" TYPE NOTATION.

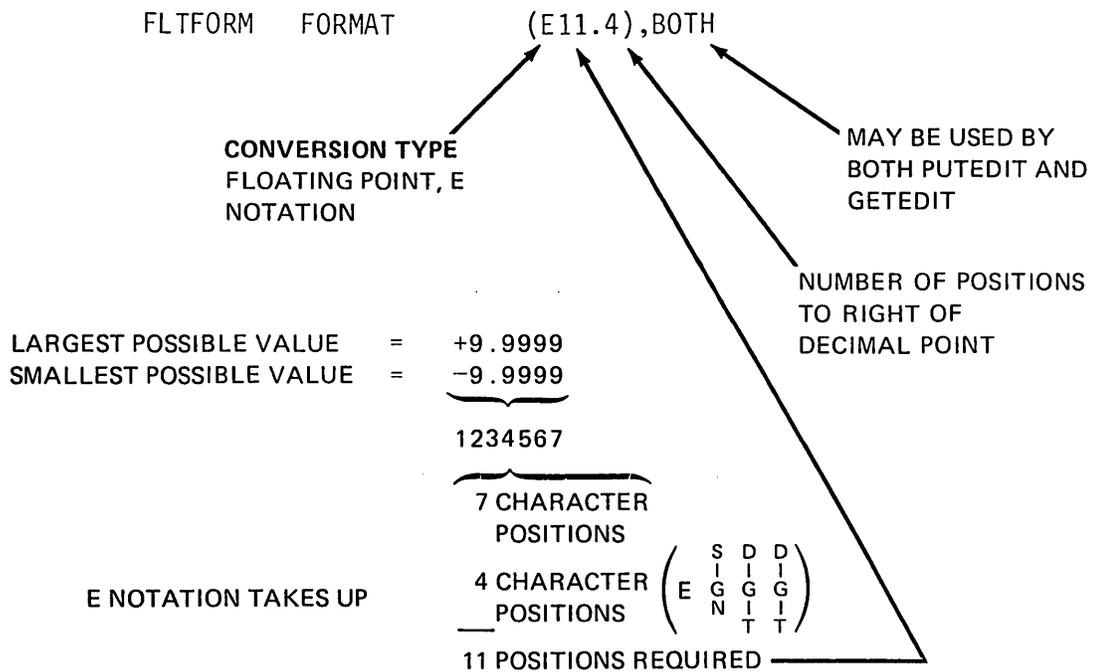


Figure 12-15. FORMAT statement E type

The second operand in the GETEDIT statement (Figure 12-16) is the location of the character buffer. The length of this buffer must be large enough to accommodate the largest character string anticipated, or truncation will result (254 characters maximum).

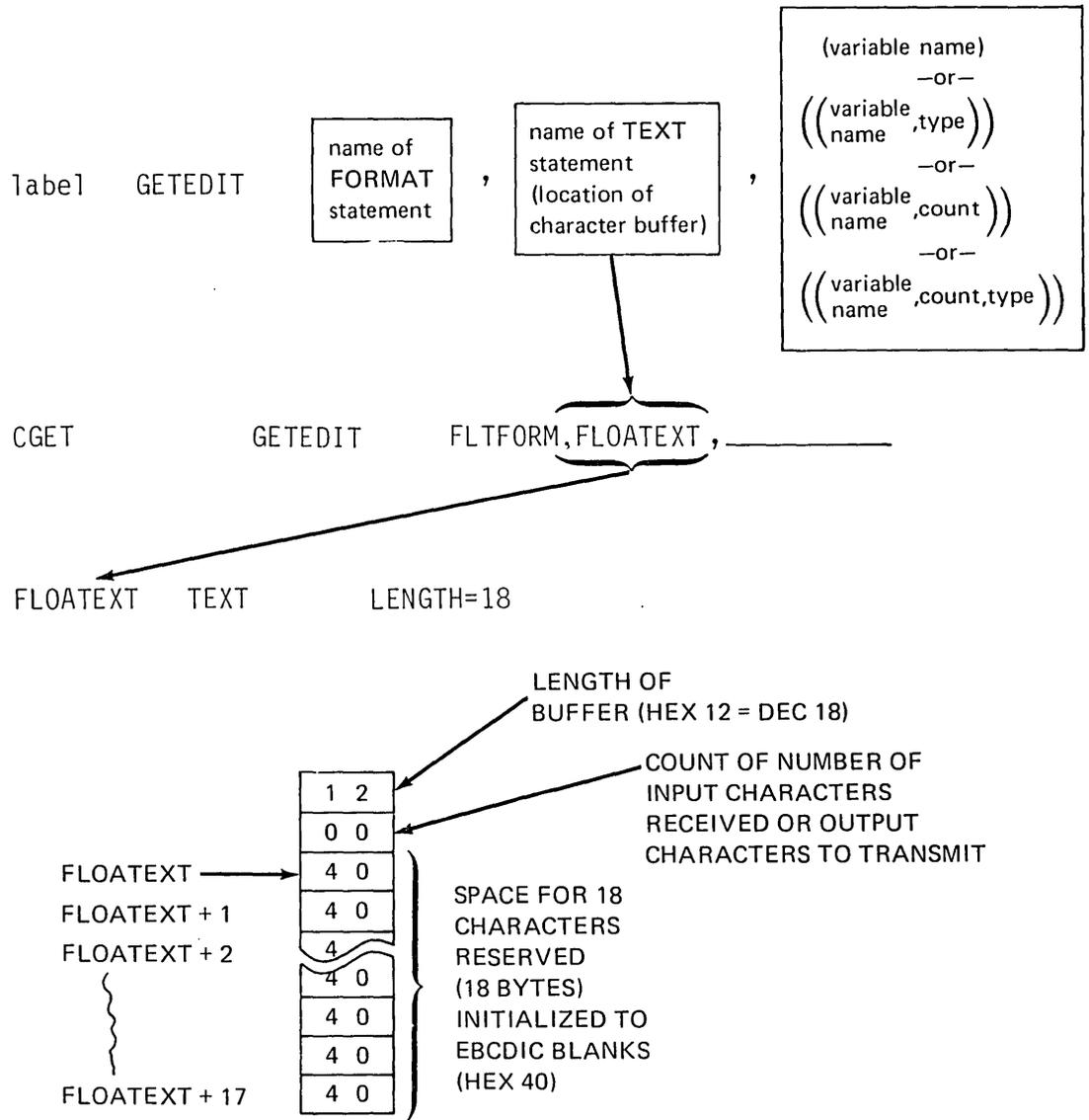


Figure 12-16. Character buffer location

Figure 12-17 summarizes the third operand, the variable list. The variable names used must previously have been defined in the program (DATA statements).

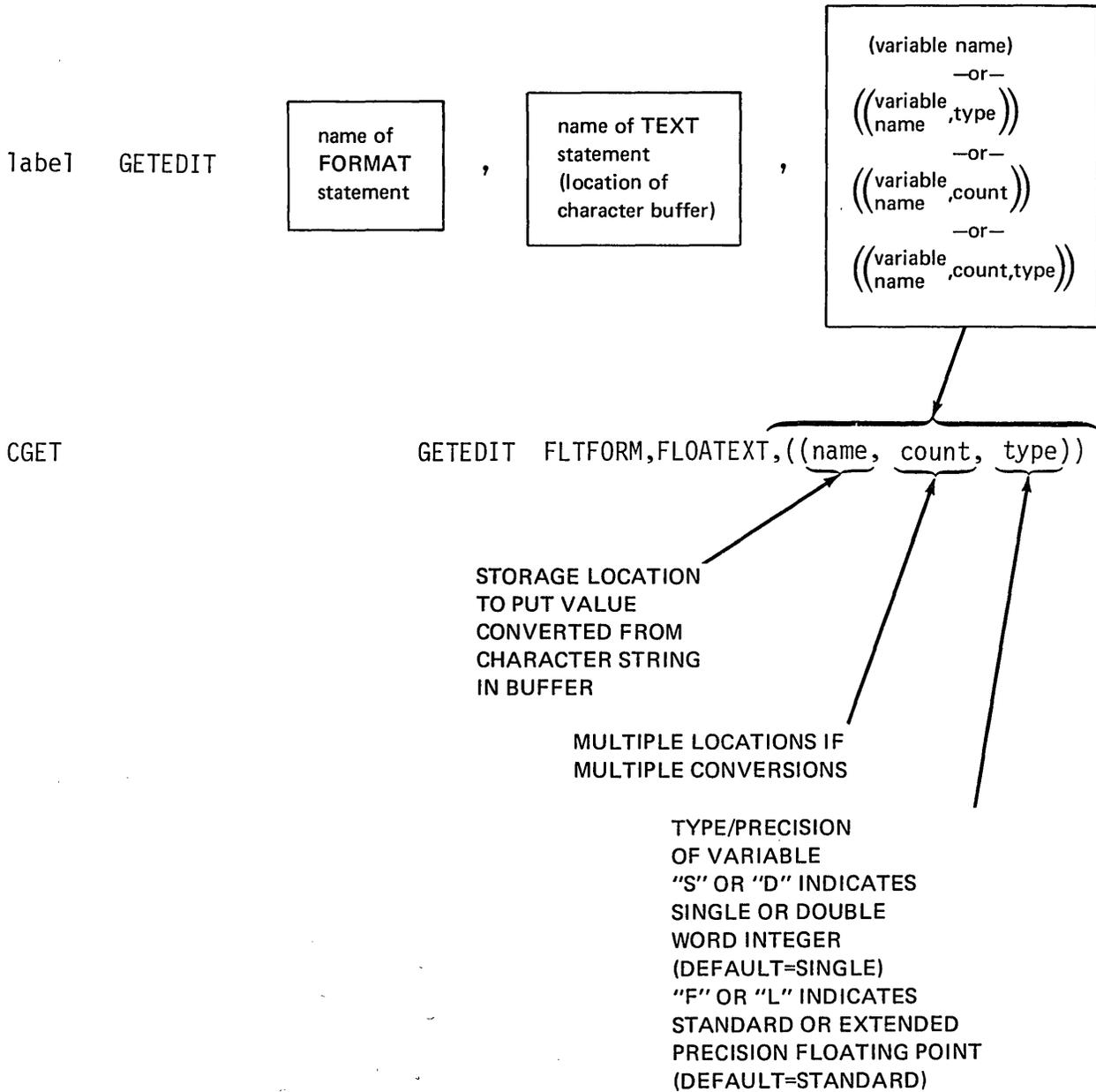


Figure 12-17. Third operand summary

If arithmetic variables are being converted, the data type specified must agree with the data conversion specification in the FORMAT statement (F or L in GETEDIT must have either F or E in FORMAT statement, and S or D in GETEDIT corresponds with I in FORMAT statement).

The completed GETEDIT statement is shown in Figure 12-18, with all three operands coded. To illustrate the optional I/O capability, a fourth operand, ACTION= is also coded. The more common usage (and the default) is ACTION=I/O, meaning a GETEDIT or PUTEDIT would implicitly issue a READTEXT or PRINTTEXT. With ACTION=STG, the GETEDIT or PUTEDIT assumes the user will take care of transferring the EBCDIC character string from or to the terminal by issuing explicit READTEXT or PRINTTEXT commands as required.

**GETEDIT**

- GETS EBCDIC CHARACTER STRING FROM A CHARACTER BUFFER SET UP BY A TEXT STATEMENT
- CONVERTS EBCDIC CHARACTER STRING ACCORDING TO SPECIFICATIONS IN FORMAT STATEMENT, AND PLACES RESULT OF CONVERSION IN STORAGE
- MAY OPTIONALLY ISSUE A READTEXT COMMAND TO TRANSFER EBCDIC CHARACTERS FROM A TERMINAL INTO THE CHARACTER BUFFER, BEFORE BEGINNING CONVERSION

**EXAMPLE:** CONVERT THE EBCDIC CHARACTER STRING IN THE CHARACTER BUFFER DEFINED BY THE TEXT STATEMENT AT LOCATION "FLOATEXT" INTO A STANDARD PRECISION FLOATING POINT NUMBER, ACCORDING TO THE SPECIFICATIONS OF THE FORMAT STATEMENT AT LOCATION "FLTFORM". STORE THE RESULT AT LOCATION "FVAL".

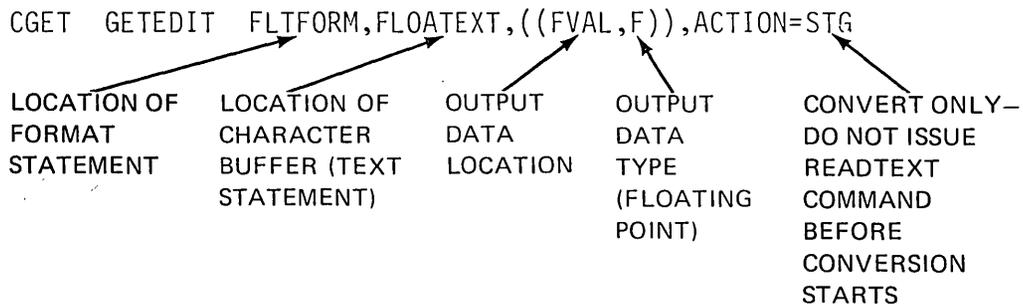


Figure 12-18. Completed GETEDIT

As a comparison, the same operation in reverse is illustrated in Figure 12-19.

**PUTEDIT**

- CONVERTS DATA IN STORAGE INTO EBCDIC CHARACTER STRING, ACCORDING TO SPECIFICATIONS IN FORMAT STATEMENT
- PLACES EBCDIC CHARACTER STRING IN CHARACTER BUFFER SET UP BY TEXT STATEMENT
- MAY OPTIONALLY ISSUE A PRINTTEXT COMMAND TO TRANSFER CONTENTS OF THE CHARACTER BUFFER TO A TERMINAL DEVICE AFTER CONVERSION

**EXAMPLE:** CONVERT THE STANDARD PRECISION FLOATING POINT VARIABLE AT STORAGE LOCATION "FVAL" INTO AN EBCDIC CHARACTER STRING, ACCORDING TO THE SPECIFICATIONS IN THE FORMAT STATEMENT AT LOCATION "FLTFORM". PLACE THE EBCDIC STRING IN THE CHARACTER BUFFER DEFINED BY THE TEXT STATEMENT AT LOCATION "FLOATEXT".

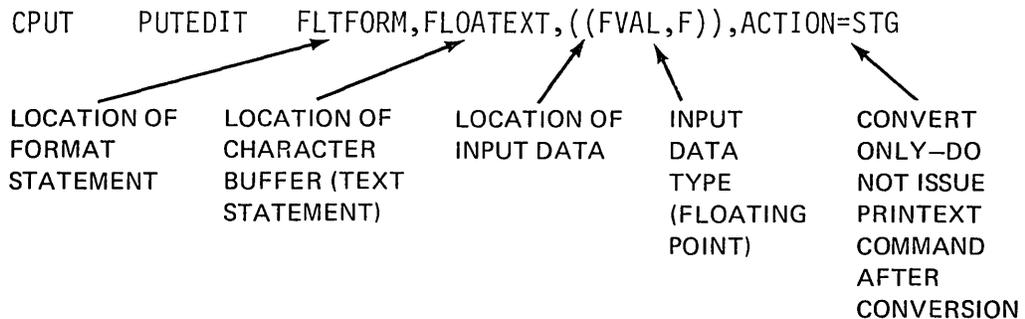


Figure 12-19. Completed PUTEDIT

All operands are in the same position, and have the same meanings for PUTEDIT as for GETEDIT; only the operation direction is reversed.

Figure 12-20 is an overview of a complete GETEDIT operation using the same examples of GETEDIT, TEXT, and FORMAT as you have seen in the previous figures. Following the numbers on the illustration, the characters entered at the terminal **1**, are transferred to the text buffer by the READTEXT instruction **2**. In this example, the READTEXT is issued by the user sometime prior to execution of the GETEDIT. If ACTION=I/O were coded in the GETEDIT (or not coded, and allowed to default), the READTEXT would be automatically issued by the GETEDIT.

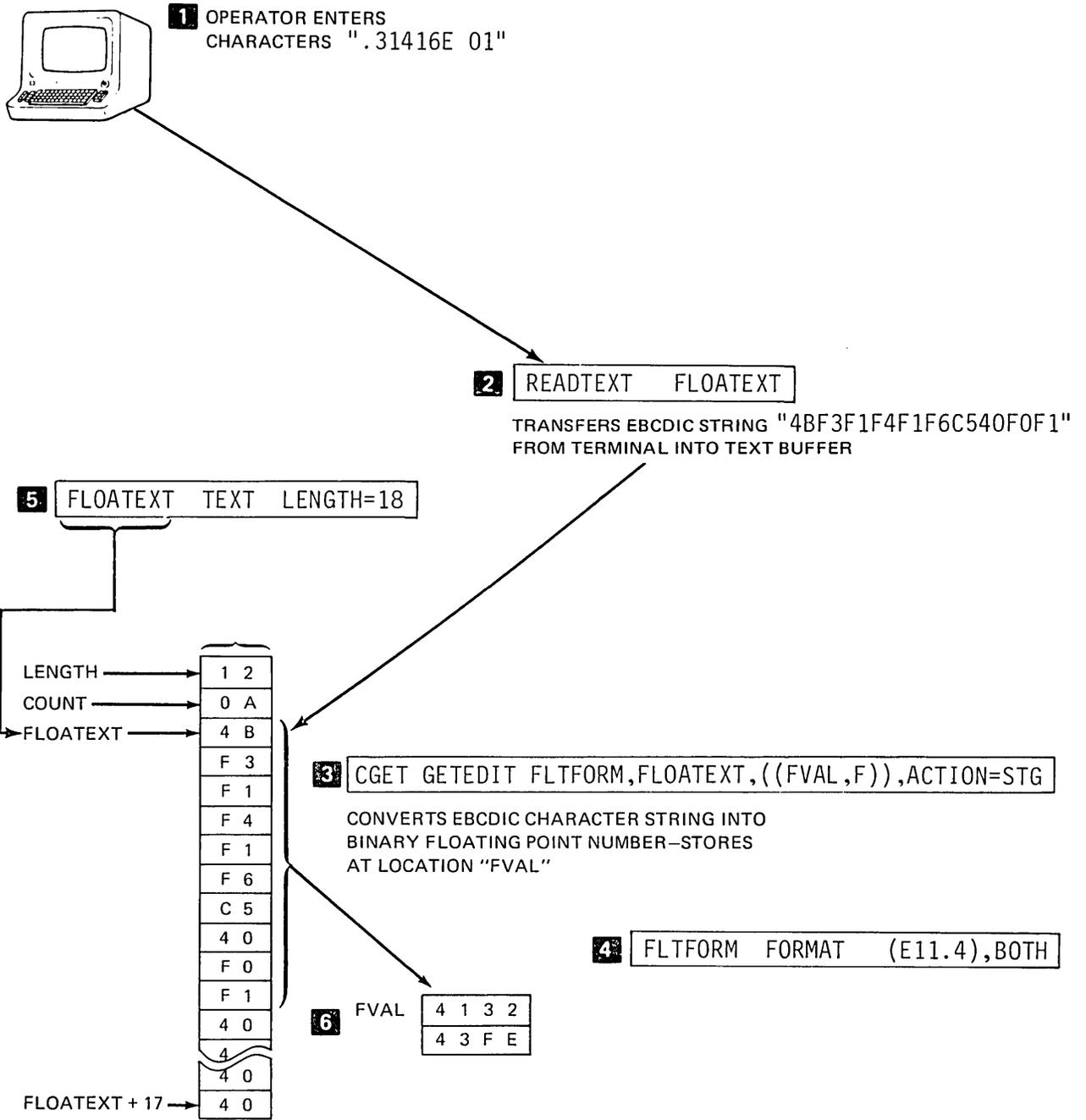


Figure 12-20. GETEDIT overview

The GETEDIT **3**, using the FORMAT statement FLTFORM **4**, converts the EBCDIC character string in the text buffer at FLOATEXT **5** into a standard precision floating point value, which is stored at FVAL **6**.

*Note:* Support for GETEDIT/PUTEDIT/FORMAT instructions is supplied in the form of object modules. When a user program containing GETEDIT/PUTEDIT/FORMAT statements is assembled, \$EDXASM automatically generates corresponding EXTRN records for use by the link edit utility \$LINK.

After an object module has been produced by \$EDXASM, it must be processed by \$LINK to include the data-formatting object modules. The user must code the AUTO= parameter in the link edit OUTPUT control statement as AUTO=\$AUTO,ASMLIB. \$AUTO is the name of a system-supplied data set on ASMLIB, which contains an autocal list, including entries for the GETEDIT/PUTEDIT/FORMAT support modules.

## DATA FORMATTING REVIEW EXERCISE—QUESTIONS

Match the instructions on the left with the statements on the right. The instructions may apply to more than one statement, and the same statement may be true for more than one instruction, or not true for any.

- |             |  |
|-------------|--|
| a. CONVTD   | 1. _____ always requires a text buffer.  |
| b. PRINTNUM | 2. _____ used to read numeric values from a terminal and convert them to internal (binary) representation. |
| c. GETEDIT  | 3. _____ may optionally perform I/O.   |
| d. CONVTB   | 4. _____ cannot be used for internal/external or external/internal conversion.                             |
| e. PRINTEXT | 5. _____ never performs I/O.   |
| f. GETVALUE | 6. _____ used to convert an EBCDIC string in a text buffer to a binary value.                              |
| g. PUTEDIT  | 7. _____ never requires a text buffer.   |
| h. READTEXT | 8. _____ always performs I/O.  |
|             | 9. _____ may be used to convert both floating point or integer values.                                     |

## DATA FORMATTING REVIEW EXERCISE—ANSWERS

1. CONVTD (a), GETEDIT (c), CONVTB (d), PUTEDIT (g), and READTEXT (h) always require a text buffer. PRINTTEXT (e) usually uses a text buffer, but may be used to issue forms control commands without any transfer of text. GETVALUE usually uses a text buffer, either implicit, as the pmsg operand, enclosed in apostrophes, or as an explicitly coded TEXT statement but may be coded without a prompt message, and therefore no text buffer.
2. GETEDIT (c) and GETVALUE (f) may be used to read numeric values from a terminal and convert them to internal (binary) representation. GETEDIT can read and convert multiple values, integer and floating point or mixed integer and floating point, of varying external format. GETVALUE can read multiple single precision integers. If the external format of the input value is other than single precision integer (double precision integer, standard or extended precision floating point in either F or E format), then the format of the input variable must be specified in the FORMAT= operand, the internal format must be specified in the TYPE= operand, and only one value can be read and converted by execution of a single GETVALUE instruction.
3. GETEDIT (c) and PUTEDIT (g) may optionally perform I/O. If the ACTION= operand is coded as ACTION=STG conversion will be performed between the internally represented variables and the text buffer specified, but no data transfer to or from a terminal will take place.
4. PRINTTEXT (e) and READTEXT (h) cannot be used for internal/external or external/internal conversion of numeric values. These two instructions deal in the transfer of text strings between storage and terminals exclusively. There may be code conversion performed, from the EBCDIC representation in a text buffer to or from whatever unique code a particular terminal requires, but this is an automatic function of the system, is transparent to the user, and is not the conversion of arithmetic values which was defined as data conversion in this section.
5. CONVTD (a) and CONVTB (d) never perform I/O. These instructions always operate between variables and text buffers in storage. All other instructions listed either always, or optionally may perform I/O.
6. CONVTD (a) and GETEDIT (c) are used to convert an EBCDIC string in a text buffer to a binary value. The GETEDIT may also have read the value into the text buffer from a terminal (ACTION=I/O).

7. PRINTNUM (b) never requires a text buffer. The conversion is from the binary value to the code required by the terminal, with no user defined text buffer employed. GETVALUE (f) does not require a text buffer for the conversion, but may use one for the prompt message if the pmsg operand is coded.
8. PRINTNUM (b), PRINTEXT (e), GETVALUE (f), and READTEXT (h) always perform I/O. I/O is optional with GETEDIT (c) and PUTEDIT (g).
9. CONVTD (a), PRINTNUM (b), GETEDIT (c), CONVTB (d), GETVALUE (f), and PUTEDIT (g), all handle single and double precision integers, and standard or extended precision floating point numbers in F or E notation external formats. PRINTEXT (e) and READTEXT (h) do not perform any conversion, and therefore do not apply.

This page intentionally left blank.

## Section 13. Sensor I/O

**OBJECTIVES:** Upon successful completion of this topic, the student should be able to:

1. Define the sensor I/O requirements in an application program.
2. Understand how to obtain digital and analog data from external devices.
3. Understand how to send digital and analog output signals from the Series/1 to external devices.
4. Use the facilities provided to service process interrupts on a Series/1.

### SENSOR BASED I/O

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive System Guide (SC34-1702), "Sensor I/O."

"Data Processing Input/Output" refers to the exchange of information between a computer and a data processing I/O device. An example of this is shown in Figure 13-1 in the form of an operator entry at a terminal, which the program in the computer then transfers into storage, and acts upon.

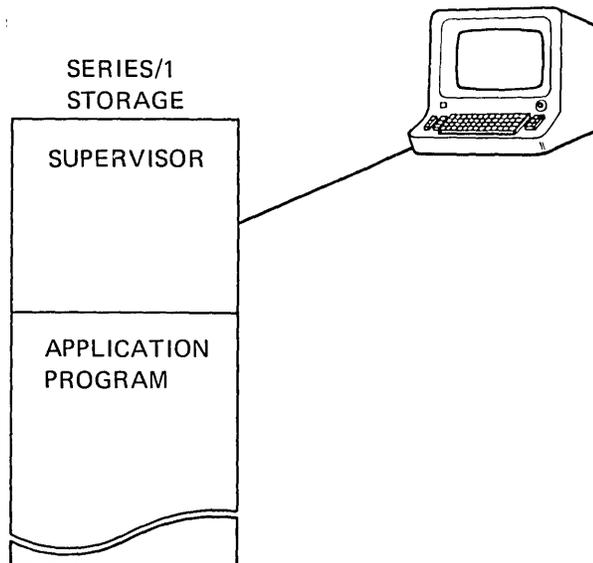


Figure 13-1. Data processing I/O

Depending on what the input means to the program, an information message or guidance prompt may be sent back to the terminal operator in response.

In Figure 13-2, the same example has been put into an applications context. Assume that the program is a "flow monitoring" application, related to some industrial process. A gauge is connected to a pipe, indicating the rate of flow through the pipe. The rate of flow can be adjusted using the valve.

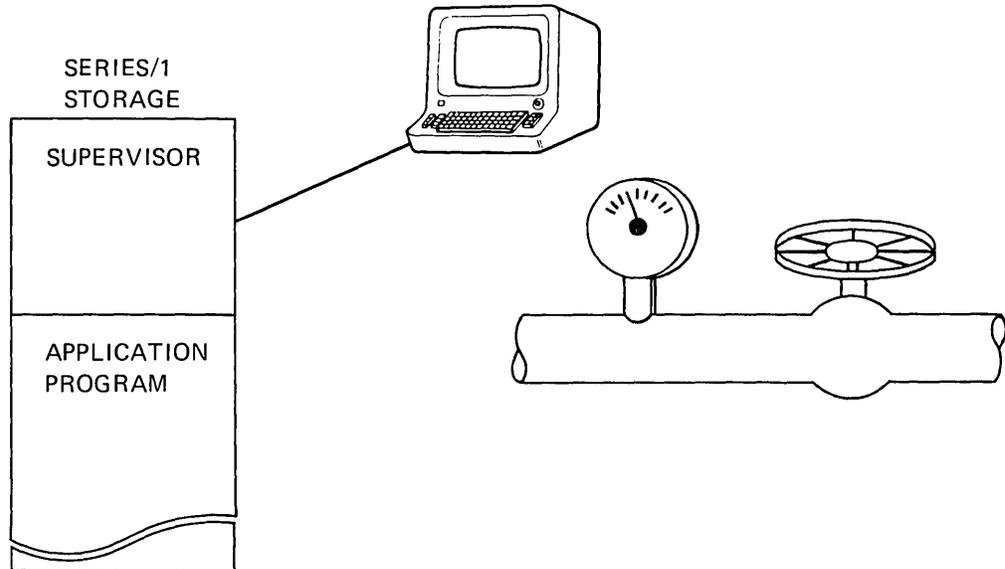


Figure 13-2. Flow monitoring

In response to a prompt from the program, the operator reads the gauge, and enters the rate of flow at the terminal. The program transfers the information into storage and checks the entered flow rate against predetermined limits or targets. If the flow rate is too high or too low, the program sends a message to the terminal instructing the operator to adjust the valve down or up.

In the example just discussed, a computer program is used to analyze a measurement of some physical property (in this case, rate of flow in pipe), and based on that analysis, request that a mechanical action take place (turn the valve up or down). The human operator, using the terminal, provided the flow rate information to the program, and as a result of a message on the terminal, provides the power to turn the valve.

Using the "Sensor Based Input/Output" features of the Series/1, the same application can be performed without using an operator or a terminal. In Figure 13-3, the gauge has been replaced by another flow-monitoring device, which translates flow rate into a voltage proportional to the rate of flow, rather than into movement of a needle around a dialface. The voltage produced is therefore an analog of the rate of flow within the pipe.

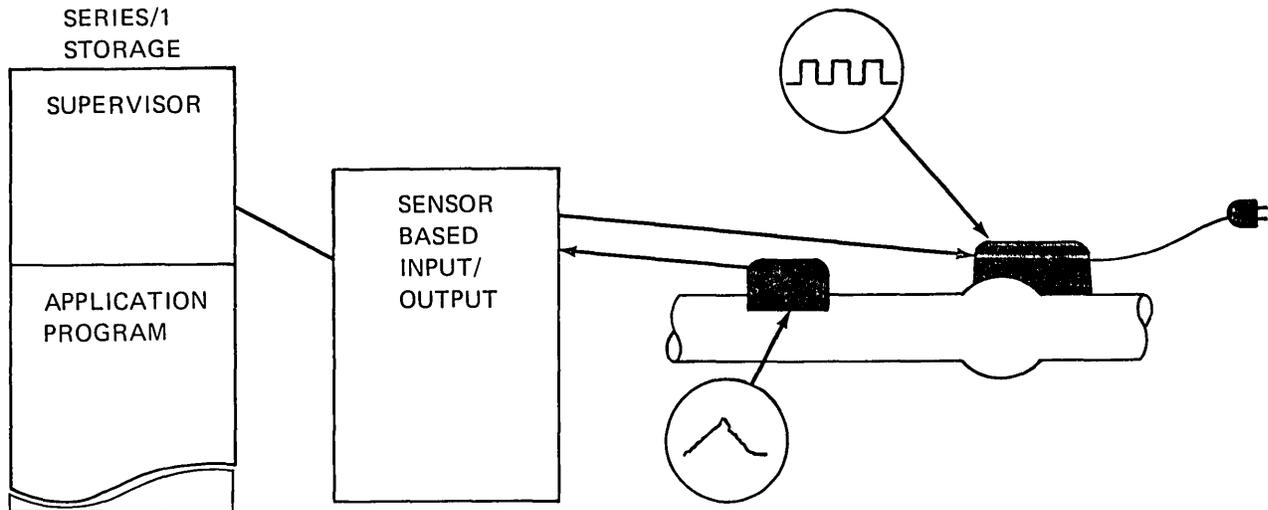


Figure 13-3. Sensor based I/O flow monitoring

The voltage is sensed by the Series/1 Analog Input (A/I) feature, and converted to a digital value (binary). This value can then be arithmetically compared with known limits or targets, and a decision can be made whether to decrease or increase the valve opening.

The manually operated valve has been replaced by a motorized unit. The direction and amount of rotation of the motor drive can be controlled by the Digital Output (D/O) sensor I/O feature.

The entire "flow-monitoring" application can now be directly controlled by the program, from acquisition of the flow-rate information (A/I), through the performance of the corrective mechanical adjustment (D/O). The delays and errors inherent in operator participation in the process no longer exist.

Sensor I/O is used in a variety of application areas, including process control, laboratory automation, and plant automation. Sensor I/O devices available on the Series/1 are as follows;

## Digital Input/Output

A digital unit of sensor I/O is a physical group of 16 contiguous points. The entire group of sixteen points is accessed as a unit at the I/O instruction level; Event Driven Executive programming support allows logical access down to the single point level. Each point of Digital Input (D/I) or Digital Output (D/O) may be operated (turned on/off) independently. D/I is usually used to acquire information from instruments which present binary-encoded output, or to monitor contact/switch status (open/closed). D/O is used to control electrically operated devices through closing relay contacts, pulsing stepping motors, etc.

Process Interrupt (P/I) is a special form of D/I. If a point of D/I changes state, and then changes state again, without an intervening READ operation from the program, the status change will be undetected. With P/I, a point changing from the off state to on generates a hardware interrupt, which is then routed, through software support, to an interrupt servicing user program which can respond to the external event which caused the interrupt. P/I is often used for monitoring critical or alarm conditions, which must be serviced quickly, and whose occurrence must not go undetected.

## Analog Input/Output

A physical unit of Analog Input (A/I) may be a group of 8 points or 16 points, depending on the type. Analog Output is installed in groups of 2 points. Each point of A/I and A/O is accessed separately, at both the I/O instruction and Event Driven Executive support level.

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 "non-digitizing" instruments. Digital Input was described as monitoring an on/off status; only one of two conditions were possible. With A/I, the intelligence is carried in the amplitude of the voltage sensed rather than in its presence or absence.

Analog input voltages are converted to corresponding binary equivalents for use by the system, by the use of an Analog to Digital (A to D) converter. Figure 13-4 is a schematic of the analog input conversion mechanism.

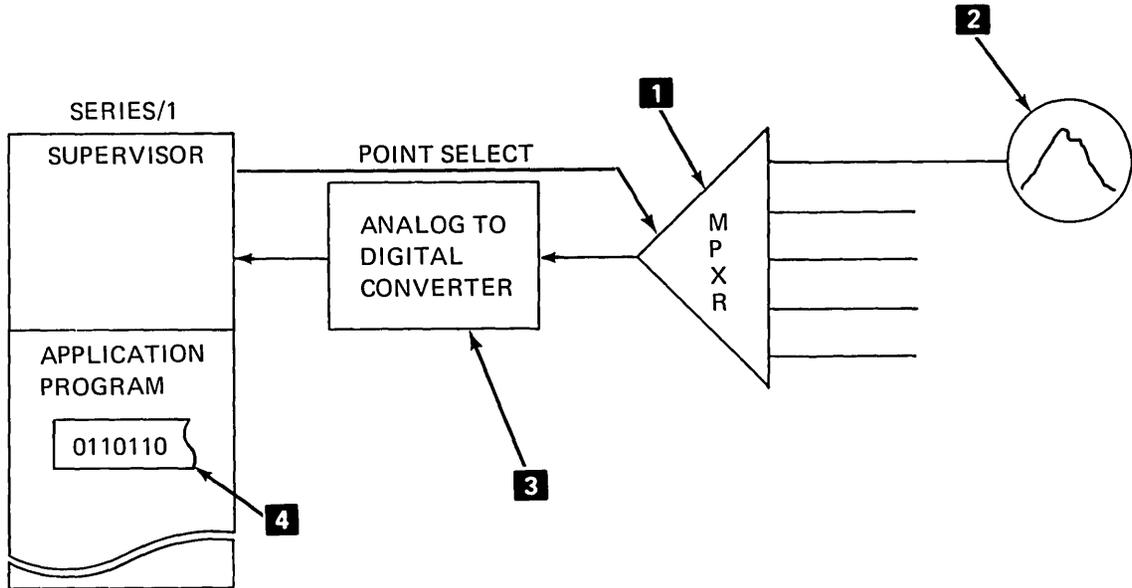


Figure 13-4. Analog to digital conversion

The address of the point to be "read" (sensed) is sent to a multiplexor **1** which selects the requested point. The voltage at the selected point **2** is routed through the multiplexor to the Analog to Digital Converter **3**. The A to D converter changes the voltage into an equivalent binary value, which can then be used in the Series/1 **4**.

With Analog Output, this process is reversed. In Figure 13-5, a binary value **1** which is the equivalent of a desired voltage, is converted to that voltage by a Digital to Analog Converter **2**, and transferred to the specified output point **3**.

For more detailed information about Series/1 Sensor I/O Features, see "IBM Series/1 4982 Sensor I/O Unit Description" (GA34-0027).

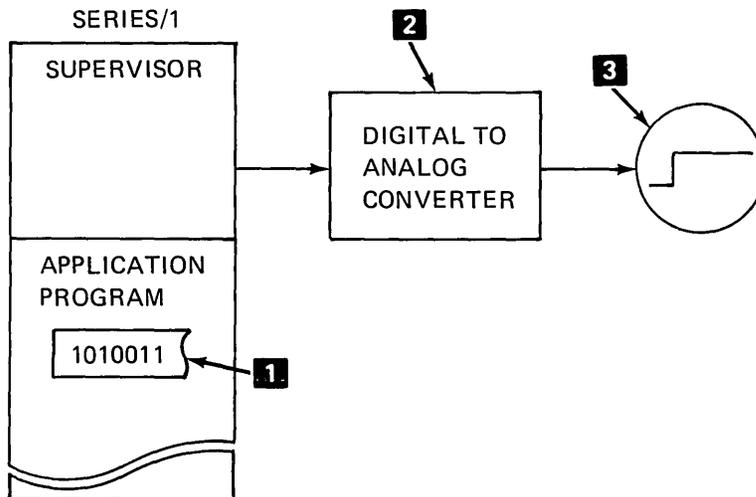


Figure 13-5. Digital to analog conversion

## EVENT DRIVEN EXECUTIVE SENSOR I/O SUPPORT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive System Guide (SC34-1702), "SENSORIO Configuration Statement."

The Event Driven Executive supplied supervisor as sent from PID contains no support for sensor I/O. If you wish to use these devices, you must do a "tailored system generation" to include the required support modules in your own supervisor. (See the "System Generation" section of this study guide for more information on generating a "tailored supervisor".)

Figure 13-6 is a graphic depiction of how sensor devices are connected to a Series/1. The devices themselves (D/I, D/O, P/I, A/O, A/I) 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 illustration, the physical connections are there, and the hardware addresses are assigned (wired in), but the supplied supervisor in storage lacks the support necessary to operate the devices.

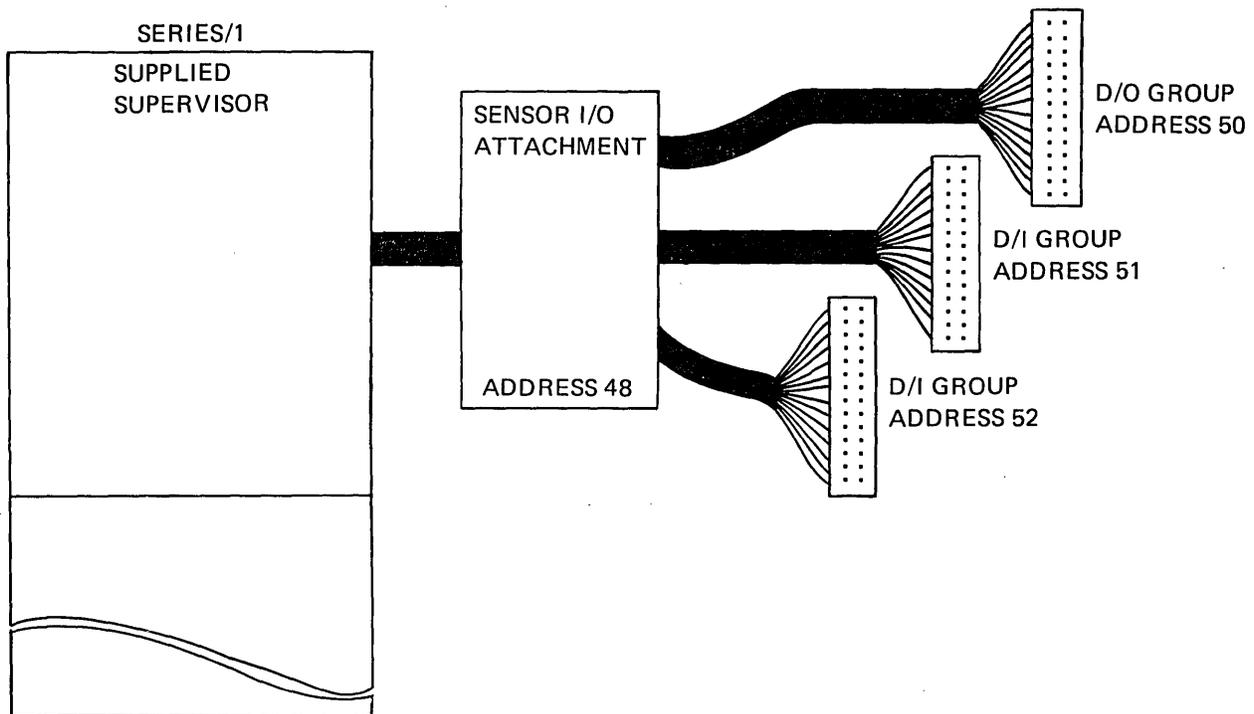


Figure 13-6. Sensor device connections

Building a "tailored supervisor" involves the assembly of a series of system configuration statements that reflect the I/O configuration and application requirements you wish to support. The system configuration statement which allows you to define sensor I/O devices is SENSORIO. Figure 13-7 illustrates the results of a tailored sysgen, using the SENSORIO system configuration statement to generate the necessary control blocks, and with sensor I/O supervisor support modules included.

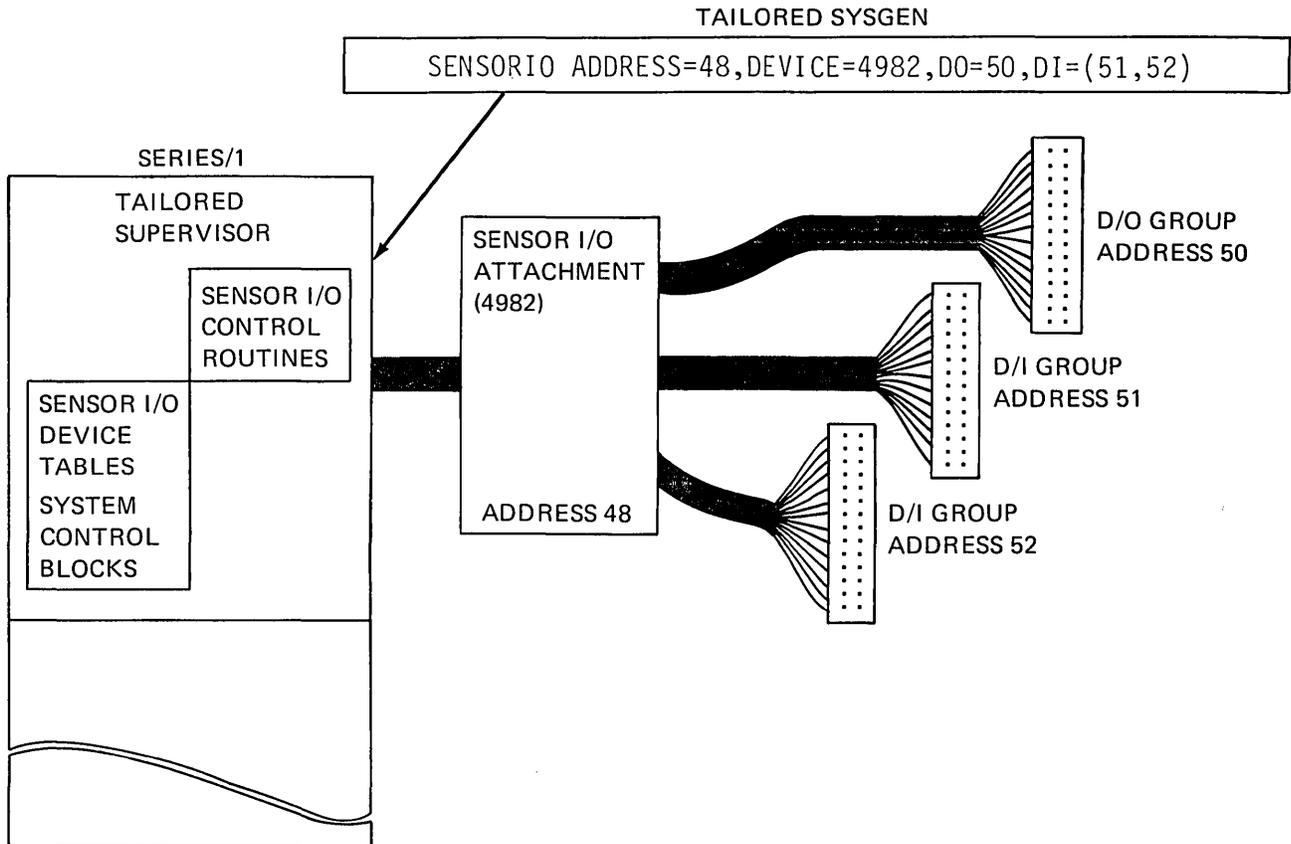


Figure 13-7. SENSORIO

## IODEF STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "IODEF."

The SENSORIO statement defined the hardware device addresses for the supervisor. When programs reference I/O devices, they use symbolic references, rather than actual addresses. The IODEF statement (I/O Definition) 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 within an application program.

Figure 13-8 illustrates an IODEF statement. Each different logical sensor device that may be used by a program must be defined in an IODEF statement. In the example, the first operand is the symbolic name of the device, "DO1". The "DO" portion of "DO1" is required, if you are defining a Digital Output device. The numeric portion may be any number you wish, from 1 through 99 (the "1" in "DO1" does not mean "1st DO device on the adapter". It is simply a symbolic reference number, used to differentiate between multiple logical devices of the same type.)

Each kind of sensor I/O is designated in the same manner; the alpha portion of the symbolic reference indicates the type of device (D/O, D/I, A/O, A/I, P/I), and the numeric portion differentiates between logical devices of the same type, and is user assigned.

The second operand in the example is coded as "TYPE=GROUP". This means that the logical digital output device, whose symbolic name is "DO1" consists of an entire group of D/O points (16 points in a group). The third operand specifies that the hardware address of this group is 50, which ties back to the hardware address for this group defined in the supervisor, during system generation.

You do not have to define a logical D/O or D/I device as consisting of all sixteen points of a hardware group. The second operand may be coded as "TYPE=SUBGROUP", in which case a fourth operand must be coded (BITS=), indicating which bit, or group of bits, within the hardware group of 16 at this address, constitutes the logical device defined by operand 1. You can therefore have multiple logical devices defined in the IODEF statement, all referencing the same physical address (group of points).

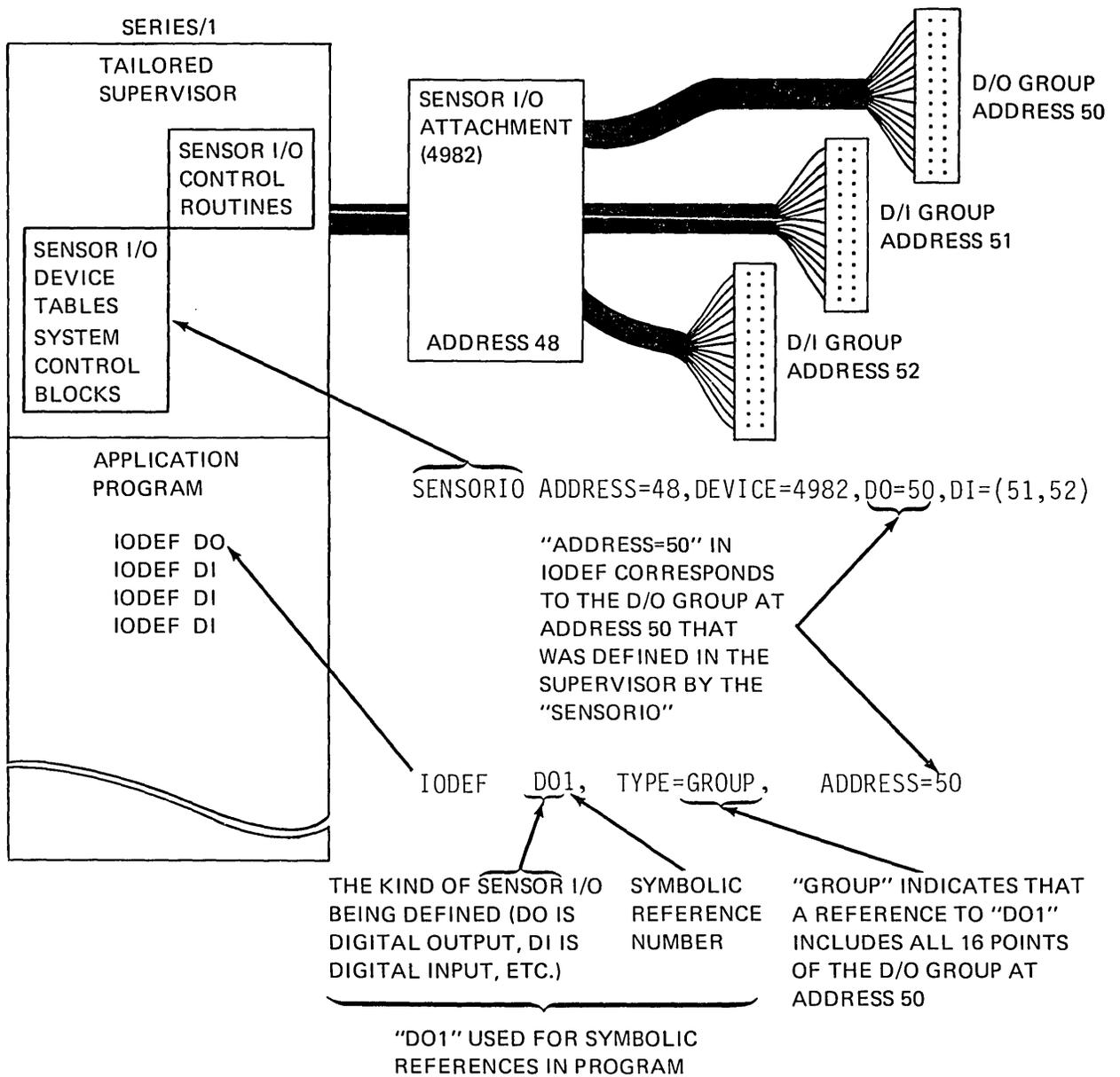


Figure 13-8. IODEF statement

## SBIO STATEMENT

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Language Reference (SC34-1706), "SBIO."

Now that the supervisor can access the hardware (SENSORIO, system generation), and you have defined the logical sensor I/O devices that will be symbolically referenced in your application program (IODEF), you are ready to do sensor I/O operations.

All sensor-based input/output operations are performed by execution of an SBIO statement. The type of operation is determined by the type of device referenced in the SBIO (READ=DI, AI, WRITE=DO, AO). In the example in Figure 13-9, the contents of location "ACON" will be written to the symbolic device "DO1", turning on the first eight digital output points of the D/O group at address 50, and turning off the second eight bits. The symbolic reference to logical device "DO1" in the SBIO statement is linked to the definition of "DO1" in the IODEF statement, which relates that device to the sixteen digital output points of hardware address 50, through the supervisor support set up at sysgen.

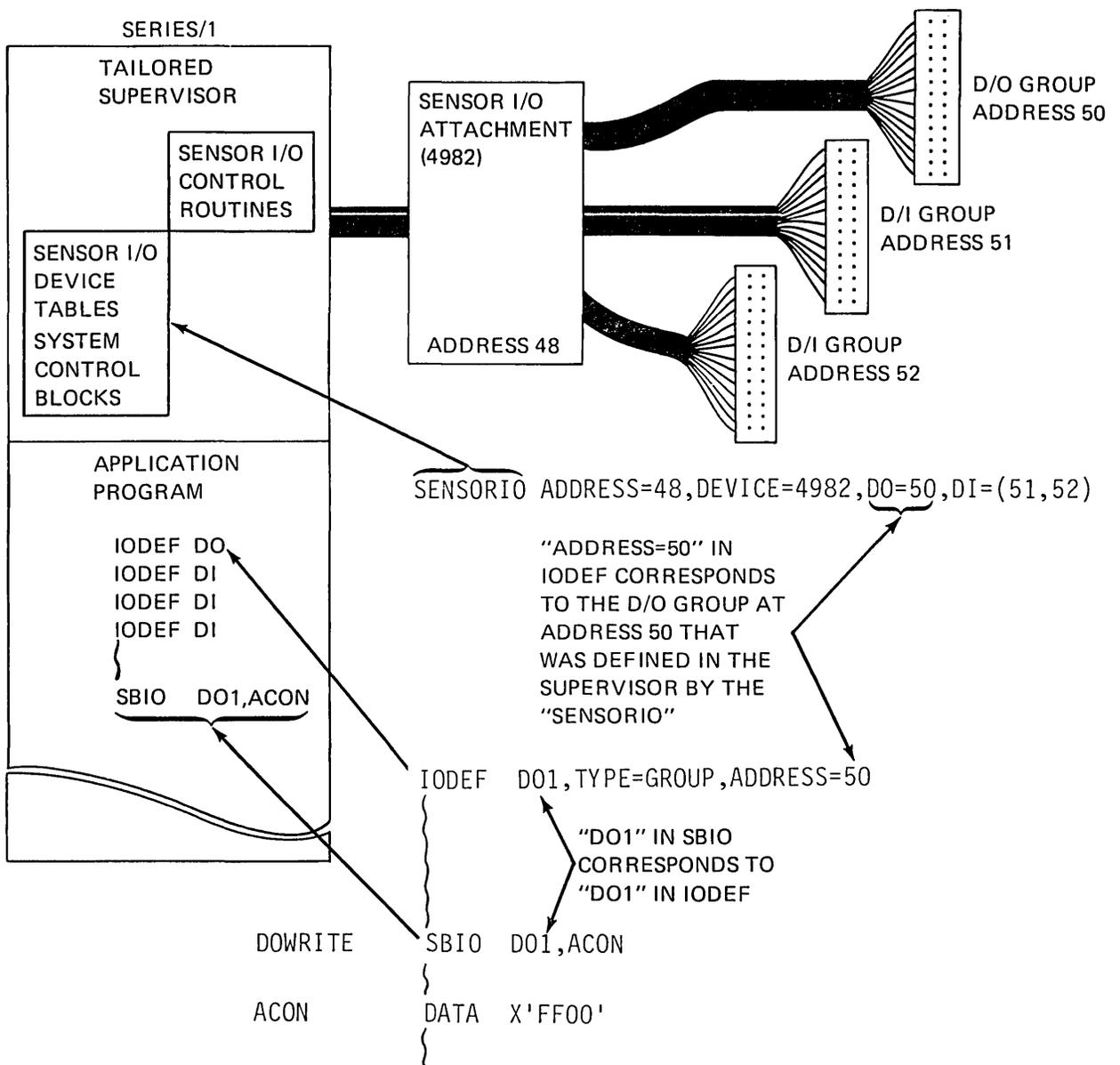


Figure 13-9. SBIO statement

## Sensor I/O Coding Examples

Following are a few IODEF/SBIO coding examples using various sensor I/O features. In all cases, assume that a tailored sysgen has been accomplished using a SENSORIO statement which supports the addresses referenced in the IODEF statements in the examples.

### *Digital Input*

```
.  
. IODEF  DI1,TYPE=SUBROUP,ADDRESS=66,BITS=(0,8)  
. SBIO   DI1,DIGIN1  
. DIGIN1 DATA  F'0'
```

**Figure 13-10. D/I example**

The IODEF defines "DI1" as being the first 8 bits of the D/I group at hardware address 66. The SBIO instruction will read these 8 bits, right justified into location "DIGIN1".





## Digital Output

Digital Output is similar to D/I in terms of coding the IODEF and SBIO instructions with one exception. D/O has the capability to send pulses, turn a D/O point on or off for a period of time, then reverse its state. This is useful in driving pulse-operated devices such as stepping motors.

```
.  
. IODEF    D01,TYPE=SUBGROUP,ADDRESS=67,BITS=(15,1)  
. IODEF  
. IODEF  
. IODEF  
. IODEF  
. SBIO     D01,(PULSE,UP)  
. SBIO  
. SBIO
```

Figure 13-13. D/O example

The above example would send a pulse to the device attached to bit 15 of the digital output group at hardware address 67. As shown, bit 15 is assumed to be off, or in the "DOWN" state when the operation begins. The "UP" says "turn bit 15 on, and then back off". "ON" may be substituted for "UP", and if going in the other direction, "OFF" may be used instead of "DOWN" when coding D/O pulse operations.

## External Sync

Both D/I and D/O may be used with external synchronization. The hardware has the capability of being "triggered" by a signal generated by a user device external to the Series/1.

```
·
·
·      IODEF    DI1,TYPE=EXTSYNC,ADDRESS=66
·
·
·
·
·      SBIO     DI1,DIWORD,1
·
·
·
·
·
·
·
·
·
·      DIWORD  DATA    F'0'
·
·
```

**Figure 13-14. External synchronization**

In the example shown above, the group at hardware address 66 will be read into location "DIWORD" only when the external synchronization signal is received.

The third operand in the SBIO statement is the number of times (count) you wish the D/I group read (how many external sync signals are to be waited for) before the supervisor posts the ECB, and execution continues.

## Analog Input

```

.
.
IODEF  AI1,ADDRESS=62,POINT=2,RANGE=5V
.
.
.
.
.
SBIO   AI1,AIVAL
.
.
.
.
.
.
AIVAL  DATA  F'0'
.
.

```

Figure 13-15. Analog input (A/I) example

The example above shows the reading and conversion of A/I point 2, defined in the IODEF as symbolic A/I device "AI1". When the conversion is complete, an I/O interrupt is generated, and the supervisor posts an ECB so that execution may continue.

The electrical value is between  $\pm 5$  volts (range). To further carry out the example, let's say the point had a value of 2.5 volts. The converted digital value in the word "AIVAL" is shown below.

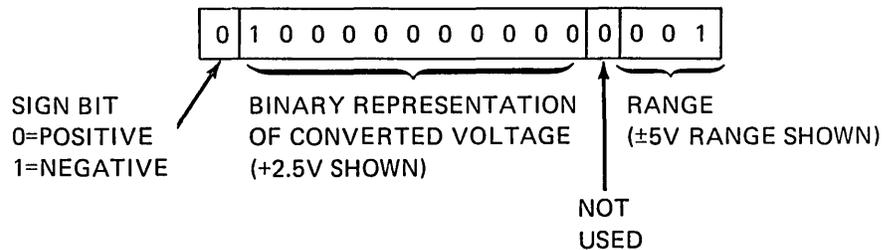


Figure 13-16. A/I conversion

For a more detailed description of A/I voltage conversion values refer to "IBM Series/1 4982 Sensor I/O Unit Description" (GA34-0027).

## Analog Output

Analog Output sends a voltage to an external user device. The program provides the binary (digital) equivalent of the desired output voltage to the A/O device, which then converts it to voltage and puts it out to the specified point.

```
.  
. IODEF      A01,ADDRESS=64,POINT=0  
. IODEF      A01,ADDRESS=64,POINT=0  
. IODEF      A01,ADDRESS=64,POINT=0  
. SBIO       A01,VOLTOUT  
. SBIO       A01,VOLTOUT  
. SBIO       A01,VOLTOUT  
VOLTOUT     DATA      X'7FC0'  
. SBIO       A01,VOLTOUT  
. SBIO       A01,VOLTOUT  
. SBIO       A01,VOLTOUT
```

Figure 13-17. Analog output (A/O) example

The above illustrates the “writing” of +5.0 volts to analog output point zero. A/O does not generate an interrupt upon completion or employ external synchronization.

The format of the output word at location “VOLTOUT” is shown below.

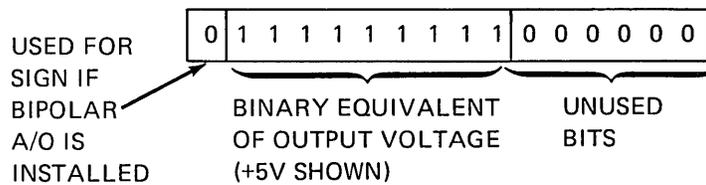


Figure 13-18. A/O conversion

For a more detailed description of A/O voltage conversion values refer to “IBM Series/1 4982 Sensor I/O Unit Description” (GA34-0027).

## SENSOR I/O REVIEW EXERCISE – QUESTIONS

1. Can a user access Sensor I/O devices executing under the Starter Supervisor? (Yes or No)
2. Using  
IODEF AI1,ADDRESS=70,POINT=2  
what will the following instruction accomplish?
  - a. SBIO AI1,TABLE,2
  - b. SBIO AI1,TABLE,2,SEQ=YES
3. Using  
IODEF DI10,ADDRESS=71,TYPE=SUBGROUP,BITS=(8,2)  
what will the following instruction do?  
SBIO DI10,DATA1
4. Using  
IODEF DO9,ADDRESS=72,TYPE=EXTSYNC  
what will the following instruction do?  
SBIO DO9,DATA,1

## SENSOR I/O REVIEW EXERCISE – ANSWERS

1. No (you must generate a tailored supervisor to access Sensor I/O).
2.
  - a. Will read AI point 2 at address 70 two times and store the converted values at the two locations at TABLE.
  - b. Will read AI points 2 and 3 once each and store the converted values at the two locations at TABLE.
3. Will read bits 8 and 9 of DI group at address 71 into storage location DATA1 (right justified)
4. Will write out the contents of storage location DATA to DO group at address 72 upon receipt of an external signal (pulse).

## Introduction to Sections 14 Through 18

The last five sections of this study guide are:

- Section 14. Utility Programs
- Section 15. System Installation
- Section 16. Program Preparation Using \$EDXASM
- Section 17. Program Preparation Using \$S1ASM
- Section 18. Session Manager

These topics address areas of the system that are most subject to change when new operating system versions are released. New utility programs may be added, or existing utilities changed or expanded in function. Almost any change to the system will also change the system installation process to some extent. Maintenance and/or enhancement of the assemblers may alter the way certain assembler functions are invoked, or change the appearance of assembler output listings.

C

C

C

## Section 14. Utility Programs

**OBJECTIVES:** Upon successful completion of this topic, the student should be able to:

1. Describe the purpose of each of the operator commands and system utility programs
2. Use the most often required utilities

### OPERATOR COMMANDS

When the ATTN key on a terminal is pressed, the system responds with the prompt character ">". An operator may then enter a character string defined in an application program's ATTNLIST statement, thereby executing a user attention routine.

There are also several operator commands that may be entered in response to the > prompt, which will cause execution of supervisor utility functions. The \$L entry is one example with which you are already familiar. \$L enqueues the system loader in preparation for loading a user or system program to storage.

Other system commands that may be entered in response to the ">" ATTN key prompt are:

#### \$A

Terminals are logically assigned or linked to particular partitions in storage, by the PART= operand of the TERMINAL system configuration statement. (For systems with < 64K of storage, all terminals are assigned to partition 1 by default.) When \$A is entered in response to the ">" prompt, the system will display the names and load points of all programs that are active within the partition to which the requesting terminal is currently assigned (see "\$CP" discussion below for how to dynamically change the partition assignment for a terminal). The command \$A ALL will display all partitions, their sizes, origins and all active programs.

## **\$B**

During normal system operation, there may be occasions when a 4978/4979/3101 Display screen becomes cluttered with residual displays from previous program executions. An example might be some protected data areas left by an application program that terminated without issuing an ERASE command. The \$B operator command will completely erase (blank) all protected and unprotected areas of the screen of the requesting terminal.

## **\$C**

This operator command is the cancel program function, and is provided as a last resort to force a program to end execution and release the storage it occupies. It is not a normal means of terminating program execution, and, depending on what the cancelled program is doing when the cancel is issued, may result in unpredictable errors. It is designed as a debug aid, and should be used with discretion.

\$C is effective only within the partition assigned to the requesting terminal. The operator will be prompted for the name of the program to be cancelled, and also for the load point, if multiple copies of the program are in execution at the same time.

## **\$D and \$P**

These two commands are on-line debug aids, which allow an operator to display (\$D) the contents of storage in hex, or to patch (\$P) storage locations from the terminal. These commands will prompt the terminal operator for starting addresses, number of words, etc., and like \$A and \$C, are effective only within the assigned partition.

## **\$CP**

The \$L, \$A, \$C, \$D, and \$P functions are all restricted to the assigned partition, as specified in the PART= operand of the TERMINAL system configuration statement defining a particular terminal. The \$CP entry is the "change partition" command, allowing dynamic reassignment of a terminal to a partition. When \$CP is entered, the operator is prompted for the number of the partition to be assigned to the terminal requesting the partition change. When the reassignment is made, all of the assigned partition only functions are effective for the new partition. See the topic "Operator Command Example" later in this section for an illustration of how the \$CP function, along with \$A and \$C, may be used.

## \$E

When system utility or application program output is directed to \$SYSPRTR, the forms are usually not advanced far enough, when the output is finished, to allow the operator to tear off the complete report. The \$E function advances \$SYSPRTR to the top of form (page eject), allowing the operator to adjust the forms position until the complete output may be removed.

## \$S

The Spooling facility provides the function of routing printer output to disk for later printing. Using the \$S operator command, a user can control the actions of the Spool facility.

## \$T and \$W

The \$T entry is the set date and time command for the 24 hour system clock/calendar. This command may only be issued from the terminals designated as \$SYSLOG or \$SYSLOGA. The date and time may be set anytime, as illustrated below.

```
> $T
```

```
DATE (M. D. Y): 10.6.78  
TIME (H. M): 13.6
```

```
DATE = 10/06/78  TIME = 13:06:36
```

Figure 14-1. \$T command

*Note:* In Figure 14-1, and in all illustrations in this section, depicting operator/utility prompt/response sequences, operator entries will be shown enclosed in boxes.

The \$W command displays the 24 hour clock and the date, and may be entered from any terminal.

```
> $W
```

```
DATE = 10/06/78  TIME = 13:06:53
```

Figure 14-2. \$W command

## \$VARYON and \$VARYOFF

The \$VARYON and \$VARYOFF commands allow a terminal operator to place tape or diskette devices in an online (\$VARYON) or offline (\$VARYOFF) status. \$VARYOFF might be useful in a situation where program testing and development are going on, and the operator wishes to make certain that production data residing on a diskette is inaccessible to the test programs.

\$VARYON is frequently used to place diskette volumes online. At system IPL, if a diskette is not mounted in the diskette drive, the diskette device is placed offline. When a diskette is mounted, or when a mounted diskette volume is removed and another volume mounted, the operator must issue a \$VARYON to place the device and volume online.

```
> $VARYON  
IODA = 02  
ASMVOL ONLINE
```

Figure 14-3. \$VARYON command

In Figure 14-3, the diskette volume ASMVOL has been mounted, and placed online with a \$VARYON command.

Notice that \$VARYOFF and \$VARYON prompt the operator for an I/O Device Address (IODA=). These commands are effective at a device level, and across the entire system.

## OPERATOR COMMAND

The following is a hypothetical situation designed to illustrate the use of the \$A, \$C, and \$CP operator commands.

We have made two assumptions:

1. A three partition Event Driven Executive system with partition 1 assigned to a 4979 (\$SYSLOG), partition 2 assigned to a 4978, and partition 3 assigned to a TTY device
2. Program debug and testing is going on in partition 1, a production job is running in partition 2, and partition 3 is currently not in use.

The application programmer using partition 1 has just produced a load module named TESTPROG, which he now wishes to test. The TESTPROG load module just produced is stored on volume EDX002. An earlier version of TESTPROG resides on volume EDX003. The programmer inadvertently loads the old version of TESTPROG, which goes into execution.

```
> $L TESTPROG,EDX003
TESTPROG      10P,13:10:27, LP = 5F00
```

Figure 14-4. 1st load

The programmer soon realizes the wrong TESTPROG has been loaded, and without terminating the program, presses the ATTN key and requests the load of the new version of TESTPROG, this time using the proper volume.

```
> $L TESTPROG,EDX002
TESTPROG      12P,13:12:00, LP = 6900
```

Figure 14-5. 2nd load

The new version of TESTPROG begins execution. The program enqueues for the loading terminal, and before a DEQT is issued, a program logic error causes an execution loop. The ATTN key produces no response, because the requesting terminal is enqueued. The programmer cannot, therefore, cancel (\$C) either TESTPROG from this terminal. If the system were re-IPLed to recover, the production job running in partition 2 would have to be terminated, a possibility that may or may not be practical.

Since the TTY device assigned to partition 3 is not in use, the programmer moves to the TTY, and wanting to know what partition it is assigned to, enters the following;

```
> $A
PROGRAMS AT 13:13:14
IN PARTITION #3  NONE
```

Figure 14-6. P3 \$A

The TTY is still assigned to partition 3, the IPL configuration specified in the TERMINAL statement defining the TTY terminal. No programs are presently active in partition 3.

The programmer now switches the TTY to partition 1, and displays the programs there.

```
> $CP
PARTITION # ? 1
> $A
PROGRAMS AT 13:14:46
IN PARTITION #1
TESTPROG 5F00
TESTPROG 6900
```

Figure 14-7. P1 \$A

Both versions of TESTPROG are displayed, along with their load points in partition 1. The next step is to cancel the looping program, freeing up the enqueued \$SYSLOG.

```
> [C]
PGM NAME: [TESTPROG]
LOAD POINT = [6900]
TESTPROG CANCELLED AT 13:15:12
> [C]
PGM NAME: [TESTPROG]
TESTPROG CANCELLED AT 13:15:59
> [SCP]
PARTITION # ? [3]
```

Figure 14-8. "\$C"

The system prompts for load point on the first cancel, because two programs of the same name are in the partition. If the first program cancelled were the one which had the 4979 enqueued, the operator could then go back to the 4979, which would now respond to the ATTN key, and terminate the remaining version of TESTPROG normally, or cancel it, if necessary. In this example, he continues with a cancel of the other TESTPROG from the TTY. Note that no load point is required when only one program of that name is active in the partition.

The TTY is then switched back to partition 3. IF this is not done, future operator commands, including \$L, issued from the TTY would still apply to partition 1.

*Note:* \$A ALL could have been used to display all partitions. \$A was used to show the use of the \$CP command.

## SYSTEM UTILITY PROGRAMS –INTRODUCTION

In this section, the Event Driven Executive utility programs are discussed under five major groups:

1. Data Management/Maintenance Utilities
2. Terminal Utilities
3. Miscellaneous Utilities
4. Program Preparation Utilities
5. Utilities supporting system facilities and features not covered in this study guide.

Groups 1 through 4 include all of the most commonly used utility programs pertaining to facilities or topics discussed in this study guide. For the most part, Event Driven Executive utilities are self-tutoring; entry of a "?" in response to a COMMAND (?): prompt will result in a display of all the command options available for that utility. Most command options are self-explanatory. Discussion of the simpler utilities will be limited to an illustration of the command options available (terminal output resulting from "?" command response), with a brief explanation of complex/obscure commands if required. Numerous examples of actual utility operation may be found in the reading assignments, and are not duplicated here.

Some utilities have specialized and/or seldom used functions. Where this is the case, utility operation is discussed and illustrated in full.

Group 4 consists of those utilities required for source program preparation. Some are covered in this section, but for most, discussion is reserved for "Section 16. Program Preparation Using \$EDXASM." Group 5 are those utilities supporting communications/graphics facilities, topics which are not addressed in this course. Discussion is limited to a brief description; no illustrations are provided, and instead of a READING ASSIGNMENT, there will be a READING REFERENCE, often in a system reference manual (SRL) not listed as one of the three manuals required for completion of this course.

## DATA MANAGEMENT/MAINTENANCE UTILITIES

### \$DASDI

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Utilities, Operator's Reference, Messages and Codes (SC34-1703), "\$DASDI Utility Program."

Before Event Driven Executive logical volumes can be defined on disk or diskette, the magnetic surface of the disk/diskette must be prepared for use. This preparation consists of a surface analysis, wherein test data is written, and then read back and checked for accuracy. If defective sectors are found, they are flagged, and alternates are assigned. Sector addresses are written, and in the case of diskettes, 128 byte sectors are established.

The Event Driven Executive utility used for disk/diskette surface preparation is \$DASDI. Figure 14-9 is the prompt/response sequence resulting from initialization of a diskette mounted on a 4966 Diskette Magazine Unit.

```

> $L $DASDI
$DASDI      7P,05:30:55, LP= 7600
DIRECT ACCESS DEVICE INITIALIZATION
DISK INITIALIZATION OPTIONS:
  0=CREATE STAND-ALONE DUMP DISKETTE 4964/4966
  1=4964, 4966 DISKETTE INITIALIZATION
  2=4962 DISK INITIALIZATION
  3=4963 DISK INITIALIZATION
  4=EXIT DISK INITIALIZATION
ENTER DISK INITIALIZATION OPTION: 1

*****
*          DISKETTE FORMATTING PROGRAM          *
* IF FORMATTING IS IN PROGRESS, DO NOT        *
* CANCEL ($C) THIS PROGRAM.  INSTEAD,        *
* ENTER ATTN/$IDSKETT TO FORCE TERMINATION.  *
*****

ENTER DISKETTE ADDRESS IN HEX 22

INITIALIZE FOR USAGE WITH THE IBM EVENT DRIVEN EXECUTIVE? Y
DEVICE VARIED OFFLINE
** WARNING **
FORMATTING WILL DESTROY ALL DATA ON THE DISKETTE IN SLOT 1. CONTINUE? Y
IBMEDX VARIED ONLINE

FORMATTING COMPLETE

LOAD $INITDSK?

```

Figure 14-9. Diskette surface preparation

The 4966 has a capacity of 23 diskettes, 2 magazines of 10 diskettes each, plus three slots for individual diskettes. The three individual slots are the first three slots in the device. \$DASDI operates on slot 1 only; any diskette on which surface preparation is to be run must first be mounted in slot 1.

After surface analysis is complete, \$DASDI writes the volume label IBMEDX, on the diskette. The next step after preparing a diskette surface is usually to create a logical volume for use with the Event Driven Executive. Logical volumes are created (directories established, etc.) with the \$INITDSK utility. \$DASDI therefore gives you the option of going directly into \$INITDSK execution, without having to end \$DASDI and issue the \$L command for \$INITDSK yourself.

Surface analysis, sector address writing, and alternate sector assignment for 4962 and 4963 disks is done at the factory. When you receive your disk unit it is ready to go; you do not have to run \$DASDI before running \$INITDSK.

While running your system (after logical volumes have been created), it is possible that a sector on disk may go bad. If this occurs, \$DASDI can be used to flag the defective sector and assign an alternate, without disturbing the rest of the data on the disk.

## \$INITDSK

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$INITDSK Utility Program."

\$INITDSK is the utility program used to establish libraries on Event Driven Executive logical volumes. Figure 14-10 illustrates the initial prompt/response sequence when \$INITDSK is loaded.

```
> $L $INITDSK
$INITDSK      58P, LP= 8C00

COMMAND (?): ?
ID - INITIALIZE DEVICE
AV - ALLOCATE VOLUME
AF - ALLOCATE FIXED HEAD VOLUME
SV - SPLIT VOLUME
IV - INITIALIZE VOLUME
II - INITIALIZE IPL TEXT
VV - VERIFY VOLUME
LV - LIST VOLUMES
DV - DELETE VOLUME
EN - END PROGRAM
```

**Figure 14-10.** \$INITDSK options

Figure 14-11 illustrates the step by step procedure of using \$INITDSK to set up a disk prior to installing an EDX System. The ID command creates a volume directory for a disk at address 03. Three volumes (EDX002, ASMLIB and EDX003) are allocated. Directories are created for each volume capable of handling 500 data sets each. A nucleus (\$EDXNUC) is allocated in EDX002 and IPL text is written pointing to \$EDXNUC as the supervisor to be loaded at IPL time.

\$INITDSK 58P, LP= 8D00

COMMAND (?):

INITIALIZE DEVICE WILL DESTROY ALL DATA  
CONTINUE?  Y  
DISK INITIALIZED  
ALLOCATE A VOLUME?  Y  
VOLUME:   
SIZE IN RECORDS:   
EDX002 ALLOCATED  
INITIALIZE THE VOLUME JUST ALLOCATED?  Y  
MAXIMUM NUMBER OF DATASETS:   
DO YOU WANT WRITE VERIFY FOR THIS VOLUME?  N  
ALLOCATE \$EDXNUC?  Y  
VOLUME INITIALIZED  
INITIALIZE IPL TEXT?  Y  
IPL TEXT WRITTEN  
ALLOCATE ANOTHER VOLUME?  Y  
VOLUME:   
SIZE IN RECORDS:   
ASMLIB ALLOCATED  
INITIALIZE THE VOLUME JUST ALLOCATED?  Y  
MAXIMUM NUMBER OF DATASETS:   
DO YOU WANT WRITE VERIFY FOR THIS VOLUME?  N  
ALLOCATE \$EDXNUC?  N  
VOLUME INITIALIZED  
ALLOCATE ANOTHER VOLUME?  Y  
VOLUME:   
SIZE IN RECORDS:   
EDX003 ALLOCATED  
INITIALIZE THE VOLUME JUST ALLOCATED?  Y  
MAXIMUM NUMBER OF DATASETS:   
DO YOU WANT WRITE VERIFY FOR THIS VOLUME?  N  
ALLOCATE \$EDXNUC?  N  
VOLUME INITIALIZED

COMMAND (?):

\$INITDSK ENDED

Figure 14-11. Using \$INITDSK

## \$DISKUT1

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$DISKUT1 Utility Program."

```
> $L $DISKUT1
$DISKUT1      37P, LP= 6300

USING VOLUME EDX002

COMMAND (?): ?
AL ---- ALLOCATE SPACE
CV ---- CHANGE VOLUME
DE ---- DELETE MEMBER
EN ---- END THE PROGRAM
LA *--- LIST ALL (DS/PGM)
LACTS * LIST ALL (CTS MODE)
LD *--- LIST DATA SETS
LDCTS * LIST DATA SETS (CTS MODE)
LM ---- LIST 1 MEMBER
LP *--- LIST PROGRAMS
LPCTS * LIST PROGRAMS (CTS MODE)
LS ---- LIST SPACE
LV *--- LIST THROUGH VOLUMES (DS/PGM)
LISTP - DIRECT LISTING TO $SYSPTR
LISTT - DIRECT LISTING TO TERMINAL
RE ---- RENAME A MEMBER
*--- PREFIX (OPTIONAL)
COMMAND (?):
```

Figure 14-12. \$DISKUT1 commands

\$DISKUT1 provides many of the most frequently required DASD storage management functions. Those functions annotated as PREFIX (OPTIONAL) indicate that if a 1 to 8 character text string is entered with the command, the command will apply to those members whose name begins with that text string.

When the utility is first loaded, commands apply to the IPL volume, but may be switched to other volumes with the Change Volume (CV) command. The only exception is the List Through Volumes (LV) function, which lists the data sets and programs in all volumes currently online, without requiring entry of CV to switch from one volume to another.

## \$DISKUT2

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$DISKUT2 Utility Program."

```
> $L $DISKUT2
$DISKUT2      46P,06:57:29, LP= 0000

USING VOLUME EDX002

COMMAND(?): ?

CD - CLEAR DATA SET
CV - CHANGE VOLUME
DP - DUMP DS OR PGM ON PRINTER
DU - DUMP DS OR PGM ON CONSOLE
    (-CA- WILL CANCEL)
PA - PATCH DS OR PGM
SS - SET PROGRAM STORAGE PARM
LP - LIST DS ON PRINTER
LU - LIST DS ON CONSOLE
PL - LIST LOG ON PRINTER
LL - LIST LOG ON CONSOLE
EN - END PROGRAM

COMMAND(?):
```

Figure 14-13. \$DISKUT2 commands

In addition to the functions listed in Figure 14-13, all of which normally operate on symbolically named data or program members, \$DISKUT2 has the capability to operate on absolute record numbers within a volume. If, when prompted for data set name, the operator enters the special system name \$\$EDXVOL, the operation will then be directed to absolute record numbers, rather than symbolic data/program member names, with record number 1 being the first record in the volume.

Similarly, if the special system name \$\$EDXLIB is entered, absolute record numbers will be used, and the first record in the directory will be considered record 1. For most volumes, \$\$EDXLIB and \$\$EDXVOL will both reference the same record, as libraries on volumes usually begin at the first record in the volume.

Figure 14-14 illustrates the use of the absolute record capability.



## \$COPY

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$COPY Utility Program."

```
> $L $COPY
$COPY          33P,06:54:21, LP= 0000

COMMAND (?): ?

CD - COPY DATA SET
CV - COPY VOLUME
RF - COPY FROM BASIC EXCHANGE
RE - COPY TO BASIC EXCHANGE
  (-CA= WILL CANCEL)
EN - END PROGRAM
COMMAND (?):
```

Figure 14-15. \$COPY commands

When using \$COPY to copy library members, the target member must already exist (allocate using \$DISKUT1), and must be of the same organization as the source member. When copying program members, the target member must be of equal or greater size than the source member. When copying data members an entire member may be copied, or only a selected number of records (partial copy) may be copied. If the entire member is to be copied, the target data member must be equal to or larger than the source. If doing a partial copy, the target member need not be as large as the source, but must have enough space following the starting target record number to accommodate the number of records being copied from the source number.

The Copy Volume (CV) command allows entire volumes to be copied, providing a volume back-up capability. A disk volume may be copied to another disk volume, a diskette volume to another diskette volume, or a diskette volume to a pre-allocated data set of appropriate size (949 records for Diskette-1, 1924 for Diskette-2) on disk. (For disk volume to diskettes, see \$MOVEVOL later in this section.) Note that copy volume operations do not add the members in a source volume to the target volume; the original contents of the target volume are replaced, including the directory.

If you have two or more 4964 Diskette Units, or a 4964 and a 4966 Diskette Magazine Unit, diskette volume copies between diskette devices are possible. If you have a single diskette drive and a disk, diskette volume copies may be performed using the following procedure:

1. Allocate a target data set on disk of 949 records (Diskette 1) or 1924 records (Diskette 2)
2. Using the CV command, copy the diskette volume to the disk data set
3. Mount the target diskette on the diskette device and vary (> \$VARYON) online
4. Using the CV command, copy the contents of the disk data set to the target diskette

If you have a single 4966 Diskette Magazine Unit and a disk, the above procedure is also recommended. Diskette volume copies between different slots of a 4966 are allowed, but are very time consuming due to the slowness of the magazine diskette selection mechanism.

\$COPY, like \$DISKUT2, has an absolute record capability, using the special system names \$\$EDXLIB and \$\$EDXVOL. This allows copying of any record relative to the beginning of a volume (\$\$EDXVOL) or relative to the beginning of a library (\$\$EDXLIB). This capability might be used, for example, when copying one diskette volume to another. The CV function of \$COPY does not copy the first cylinder on diskette. If the source diskette were an IPL volume (has IPL text and \$EDXNUC), the IPL text, contained in the first record of the first cylinder, would not be copied to the target diskette, and the target diskette volume, although containing a supervisor in \$EDXNUC, would not be able to load the supervisor when the IPL key was pressed.

To copy the IPL text to the target diskette, the CD function of \$COPY can be used, with \$\$EDXVOL specified as the data set name, and record 1 specified as the first and last record to be copied.

## \$COPYUT1

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$COPYUT1 Utility Program."

\$COPYUT1 is used extensively in system installation and maintenance activities, such as installing the various system libraries received from PID, and in applying PTF fixes distributed on diskette, \$COPYUT1 differs from \$COPY in that operations are performed on entire members only; partial copies are not allowed, nor are volume copies or absolute record operation.

```
CM---COPY MEMBER FROM SOURCE TO TARGET
----- MULTIPLE COPY COMMANDS-----
CALL--COPY ALL MEMBERS FROM SOURCE TO TARGET
CAD---COPY ALL DATA MEMBERS FROM SOURCE TO TARGET
CAP---COPY ALL PROGRAMS FROM SOURCE TO TARGET
CG----COPY ALL MEMBERS STARTING WITH TEXT FROM SOURCE TO TARGET
CNG---COPY ALL MEMBERS NOT STARTING WITH TEXT FROM SOURCE TO TARGET
----- END OF MULTIPLE COPY COMMANDS-----
SQ----SET PROMPT MODE FOR ALL MULTIPLE COPY COMMANDS
NQ----RESET PROMPT MODE FOR ALL MULTIPLE COPY COMMANDS
--CA-- WILL CANCEL MULTIPLE COPY COMMANDS
CV---CHANGE SOURCE AND TARGET VOLUMES
ROLLON -SET SCREEN = NO PAUSE
ROLLOFF -RESTORE PAUSE CHARACTERISTICS
EN---END PROGRAM
? ---HELP
```

**Figure 14-16. \$COPYUT1 commands**

This utility will copy data or program members from a source volume to a target volume, and:

1. Will delete a member from a target volume, if a member exists with the same name as the member being copied from the source volume
2. Will allocate a member on the target volume of the same size and data organization as the source member – preallocation not required
3. Will copy multiple members with a single command (all, all data, all program, generic, non-generic), with or without a prompting pause

```

> $L $COPYUT1
$COPYUT1 48P,00:00:14, LP= 6000

***WARNING MEMBERS ON TARGET VOLUME WILL BE OVERWRITTEN***

THE DEFINED SOURCE VOLUME IS EDX002, OK? NO
ENTER NEW SOURCE VOLUME: EDX003
THE DEFINED TARGET VOLUME IS EDX002, OK? YES
MEMBER WILL BE COPIED FROM EDX003 TO EDX002 OK? YES

COMMAND (?): CG

ENTER GENERIC TEXT: AREC
ARECPGM1 COPY COMPLETE      300 RECORDS COPIED
ARECPGM5 COPY COMPLETE      100 RECORDS COPIED
ARECPGM3 COPY COMPLETE      50 RECORDS COPIED

COMMAND (?):

```

**Figure 14-17. Generic copy**

Figure 14-17 is an example of a generic copy without a prompting pause. The warning message indicates that existing members with the same name as any of those being copied will be deleted.

## \$MOVEVOL

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$MOVEVOL Utility Program."

\$MOVEVOL is a dump/restore utility, used to dump entire disk volumes to diskette or restore disk volumes from diskette, where the volumes may span several diskettes.

A dumped volume consists of a control diskette containing the volume directory and control information, and as many data diskettes as are required to hold the rest of the information in the volume. See the reading assignment for information on creating the control and data diskettes, and for examples of \$MOVEVOL operation.

## \$COMPRES

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$COMPRES Utility Program."

During normal system usage, data sets in Event Driven Executive volumes will be deleted, leaving "holes" (free space) between members. The \$COMPRES utility consolidates all available free space within an Event Driven Executive volume into one contiguous area.

```
> $L COMPRES  
$COMPRES 17P,00:32:48, LP= 6900
```

```
COMPRESS SYSTEM LIBRARY  
WARNING! SHOULD BE RUN ONLY WHEN  
NO OTHER PROGRAMS ARE ACTIVE
```

```
VOLUME LABEL = EDX003
```

```
COMPRESS LIBRARY ON EDX003? YES
```

```
DIRECTORY HAS BEEN SORTED BY MEMBER IN ASCENDING ORDER.
```

```
$EDXNUC COPIED
```

```
DATA1 COPIED
```

```
PROG1 COPIED
```

```
PROG2 COPIED
```

```
THE LIBRARY IS COMPRESSED.
```

```
ANOTHER VOLUME? NO
```

```
$COMPRES ENDED AT 00:34:39
```

Figure 14-18. \$COMPRES example

The example shows compressing the members in volume EDX003. Never compress a volume when any other program is active. You can determine what programs are active by using the \$A operator command. If the compress was performed on the volume that contained the supervisor you IPLed from, and if the \$LOADER program's location was changed, you must re-IPL.

## \$TAPEUT1

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$TAPEUT1 Utility Program."

```
> $L $TAPEUT1
$TAPEUT1      20P,01:21:58, LP= 0000
```

```
COMMAND (?): [?]
```

```
CD ---- COPY TAPE DATASET
CT ---- CHANGE TAPE ATTRIBUTES
DP ---- DUMP TAPE
EN ---- END $TAPEUT1
EX ---- EXERCISE TAPE
IT ---- INITIALIZE TAPE
LT ---- LIST TAPE DRIVES AND ATTRIBUTES
MT ---- MOVE TAPE
RT ---- RESTORE DISK/VOL FROM TAPE
ST ---- SAVE DISK/VOL ON TAPE
TA ---- ALLOCATE TAPE DATASET
```

```
COMMAND (?):
```

Figure 14-19. \$TAPEUT1 options

\$TAPEUT1 provides many of the most frequently required tape management functions. When invoked, the utility displays information about tapes defined to the system. The functions provided allow a user to initialize tapes, allocate and copy tape data sets, and print tape records as well as providing a save/restore facility for disk devices.

## TERMINAL I/O UTILITIES

### \$TERMUT1

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$TERMUT1 Utility Program."

This is a general purpose terminal utility, used to perform several terminal-related functions.

```
> $L $TERMUT1
$TERMUT1 19P,01:12:29, LP= 5F00
```

```
*** TERMINAL CONFIGURATOR ***
```

```
COMMAND(?): [?]
```

```
LA -- LIST TERMINAL ASSIGNMENTS
RE -- RENAME
RA -- REASSIGN ADDRESS
RH -- REASSIGN HARDCOPY
CT -- CONFIGURE TERMINAL
EN -- END PROGRAM
```

```
COMMAND(?):
```

Figure 14-20. \$TERMUT1 options

The current terminal name, hardware address, and terminal type may be displayed using the LA (list assignment) function.

```
> $L $TERMUT1
$TERMUT1 19P,03:56:41, LP= 0000
```

```
*** TERMINAL CONFIGURATOR ***
```

```
COMMAND(?): LA
```

	NAME	ADDRESS	TYPE	PARTITION	HARDCOPY
==>	\$SYSLOG	04	4979	2	\$SYSPRTR
	DSPLY1	06	4978	3	\$SYSPRTR
	\$SYSLOGA	00	TTY	1	
	\$SYSLOGH	08	ACC1	2	(3101 BLOCK MODE)
	LINEPRTR	21	4973	1	
	\$SYSPRTR	01	4974	1	

```
COMMAND(?): EN
```

```
$TERMUT1 ENDED AT 03:57:24
```

Figure 14-21. LA

Terminals may be renamed, using the RE function. For instance, if the 4973 printer in Figure 14-21 were mistakenly referenced (ENQT) in a program as LINPRNTR the name could be temporarily changed from LINEPRTR to LINPRNTR to test the program, and then changed back.

```
COMMAND(?): [RE]
```

```
OLD, NEW TERMINAL NAMES: [LINEPRTR LINPRNTR]
```

```
COMMAND(?):
```

Figure 14-22. RE

Terminal hardware addresses (RA), hardcopy device/hardcopy PF key designations (RH), and page format configuration parameters (CT) may all be reassigned using \$TERMUT1. Reassignments remain in effect until reassigned again, or until the next IPL, which will cause all terminals to revert to the assignments in the TERMINAL system configuration statements.

## \$TERMUT2

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$TERMUT2 Utility Program."

4978 Displays have a control store and an image store, which are loaded from disk or diskette data sets. At IPL, the system automatically loads all 4978s with the control store data set \$4978CS0 and the image store data set \$4978IS0. These may be the standard system-supplied data sets, or may be user-created control/image store data sets that have been renamed \$4978CS0 or \$4978IS0.

```
> EL $TERMUT2
$TERMUT2      34P,06:58:56, LP= 0000

COMMAND (?): ?
  AD - ASSIGN DEFINE KEY
  C  - CHANGE KEY DEFINITION
  E  - END PROGRAM
  LC - LOAD CONTROL STORE
  LI - LOAD IMAGE STORE
  SC - SAVE CONTROL STORE
  SI - SAVE IMAGE STORE
  RE - RESTORE 4974 TO STD. 64 CHAR. SET
```

COMMAND (?):

Figure 14-23. \$TERMUT2 options

After IPL, \$TERMUT2 can be used to load a control or image store from user-defined control/image data sets (LC and LI commands), or to read the control or image store in a display, and write it to a user-allocated data set (SC and SI commands). Control store data sets require 16 records, and image store data sets, 8 records.

The 4978 hardware supports the DEFINE function, which allows keys to be defined with special character strings that have meaning to a particular application or job. In order to define a key with special characters, DEFINE mode must be entered. This is accomplished by pressing the DEFINE key on the 4978 keyboard.

Assuming a standard 4978 keyboard is installed, the \$4978CS0 control store supports the keyboard shown in Figure 14-24. (The unshaded keys are those that will produce hardware interrupts.)

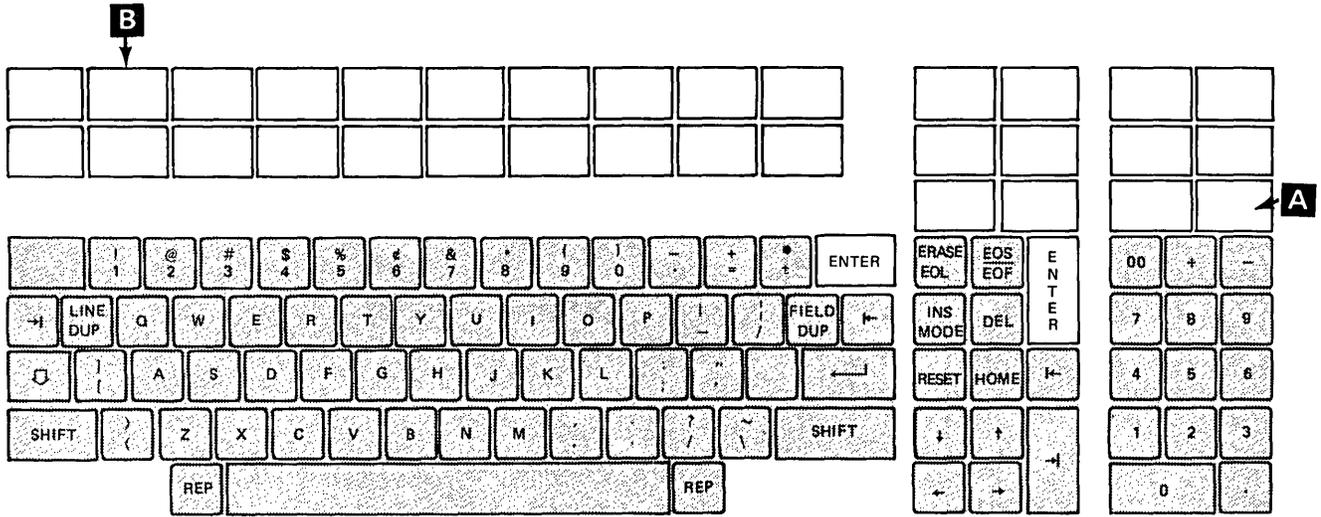


Figure 14-24. 4978 keyboard, RPO D02056

As can be seen, there is no key permanently designated as the DEFINE key. However, using the AD command of \$TERMUT2, you may assign a key of your choice as the DEFINE key.

Figures 14-25 and 14-26 are taken from the General Information manual for the 4978 keyboard (RPO D02056). Similar charts are in the General Information manuals for whichever keyboard you have installed.

In Figure 14-25, each key position is assigned a reference number. Figure 14-26 is the first page of several which list the hex scan code, function ID code, local function code, and interrupt code which comprises the control store information for each key. The identifying numbers on the keys in Figure 14-25 correspond to the key position numbers on the chart in Figure 14-26.

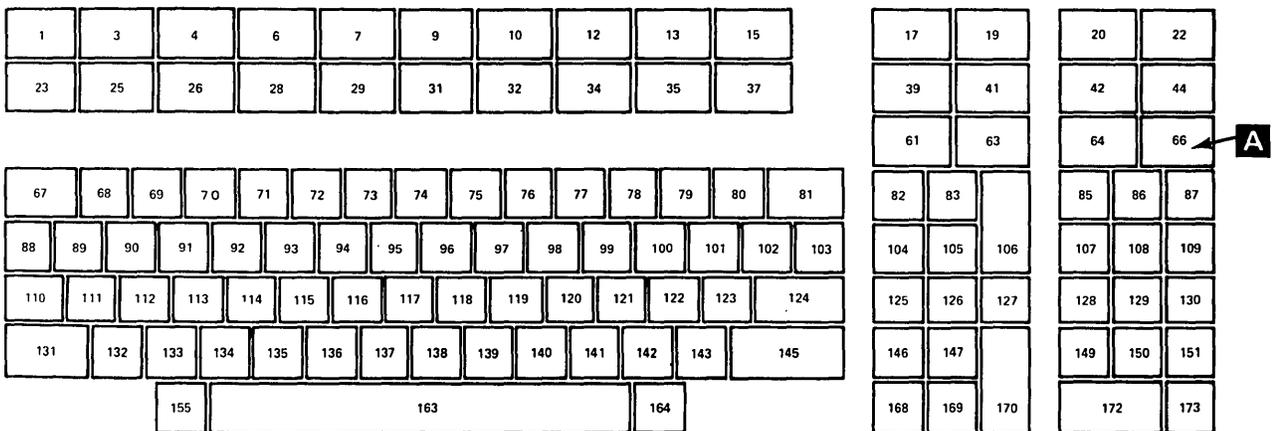


Figure 14-25. Keyboard reference assignments

Control Store Data																			
Downshift - Unshifted										Upshift - Shifted									
Key position	Scan code				Keytop symbol	Key position	Scan code				Keytop symbol								
	Function ID code						Function ID code												
	Character/local function code						Character/local function code												
	Interrupt code						Interrupt code												
	Character image table						Character image table												
	Row						Row												
	0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7		
1	01	20	00	01															
3	02	20	00	02															
4	03	20	00	03															
6	04	20	00	04															
7	05	20	00	05															
9	06	20	00	06															
10	07	20	00	07															
12	08	20	00	0B															
13	09	20	00	0C															
15	0A	20	00	0D															
17	0B	20	00	0E															
19	0C	20	00	0F															
20	0D	20	00	10															
22	0E	20	00	11															
23	0F	20	00	12															
25	10	20	00	13															
26	11	20	00	14															
28	12	20	00	15															
29	13	20	00	16															
31	14	20	00	17															
32	15	20	00	18															
34	16	20	00	19															
35	17	20	00	1A															
37	18	20	00	1B															
39	19	20	00	1C															
41	1A	20	00	1D															
42	1B	20	00	1E															
44	1C	20	00	1F															
61	1D	20	00	20															
63	1E	20	00	21															
64	1F	20	00	22															
66	20	20	00	23															
67	21	70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	(Blank)	(Blank)
68	22	00	F1	00	02	06	02	02	02	02	07	00	1					!	!
69	23	00	F2	00	07	48	01	30	04	40	78	00	2					@	@
70	24	00	F3	00	78	01	10	31	08	48	07	00	3					#	#
71	25	00	F4	00	28	28	0C	48	78	08	08	00	4					\$	\$
72	26	00	F5	00	78	40	71	08	08	48	07	00	5					%	%
73	27	00	F6	00	02	20	04	47	48	48	07	00	6					†	†
74	28	00	F7	00	78	01	10	02	20	04	40	00	7					&	&
75	29	00	F8	00	30	05	30	05	48	48	07	00	8					*	*
76	2A	00	F9	00	07	48	48	0F	01	10	02	00	9					(	(
77	2B	00	F0	00	30	05	48	4A	48	05	30	00	0					)	)

Figure 14-26. Control store data

In Figure 14-24, assume you want to make the key at **A** the DEFINE key. In Figure 14-25, that key position has a reference number of 66. In Figure 14-26, the operator is prompted for the scan code of the key to be assigned as the DEFINE key. On Figure 14-26, the scan code for key position 66 is hex 20. After the scan code and terminal name have been entered (Figure 14-27), \$TERMUT2 reloads the control store of the display, with key position 66 assigned as the DEFINE key.

```
COMMAND (?): A
  ENTER SCAN CODE OF THE KEY TO BE ASSIGNED
    AS THE DEFINE KEY (HEX): 20
  ENTER TERMINAL NAME (CR OR * = THIS ONE): DSPLY1
```

Figure 14-27. AD command

Back on Figure 14-24, the operator presses the DEFINE key at **A** key is the key which will be redefined. Assume the operator wishes to redefine Program Function Key 1, and presses it (**B** on Figure 14-23). Now all key depressions, until the DEFINE key is again depressed, will be assigned to PF1.

The operator enters the character string \$L \$FSEDIT EDITWORK, and then presses one of the two ENTER keys. He or she then presses the DEFINE key again, ending the redefinition of PF1, and taking the 4978 out of DEFINE mode.

The character string entered is a request to load the text editing utility program \$FSEDIT, along with the name of a text edit work data set, EDITWORK.

Counting the depression of the ATTN key required to get the > prompt, and the ENTER key depression following the load request, this line of text normally takes 21 keystrokes to enter into the system. Now that PF1 has been redefined as this line of text, only two keystrokes are required; the ATTN key, resulting in the > prompt, followed by PF1, which enters \$L \$FSEDIT EDITWORK and the ENTER key, which was also part of the redefinition string.

For normal terminal usage, an active DEFINE key is not desirable. If it is depressed inadvertently, altering of the control store will result. In Figure 14-28, the C command is used to change key position 66 back to its original control store configuration, using the chart in Figure 14-26 to supply the codes.

```
COMMAND (?): C
  ENTER TERMINAL NAME (CR OR * = THIS ONE): DSPLY1
  ENTER SCAN CODE OF THE KEY TO BE REDEFINED (HEX): 20
  ENTER FUNCTION ID (HEX): 20
  ENTER CHARACTER/FUNCTION CODE (HEX): 00
  ENTER INTERRUPT CODE (HEX): 23
  ANOTHER KEY? N
```

Figure 14-28. C command

At the conclusion of the C operation, the control store of 4978 DSPLY1 still has PF1 defined with the text editor load request character string, but with no DEFINE key designated. The SC operation in Figure 14-29 reads the control store, and stores it in a 16 record data set named 4978EDIT, which must be preallocated. Any time a user desires a keyboard with PF1 redesignated as a text editor load request, the LC command of \$TERMUT2 can be used to load the control store from 4978EDIT.

```
COMMAND (?): [SC]
SAVE DATA SET (NAME,VOLUME): [4978EDIT]
ENTER TERMINAL NAME (CR OR * = THIS ONE): [DSPLY1]
```

```
COMMAND (?): [END]
```

```
$TERMUT2 ENDED AT 01:27:44
```

Figure 14-29. SC command

### \$TERMUT3

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$TERMUT3 Utility Program."

\$TERMUT3 is used to enter a text message and send it to another named terminal. See the reading assignment for examples and operating instructions.

### \$PFMAP

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$PFMAP Utility Program."

When a WAIT KEY operation is terminated by pressing a Program Function key, an identifying code for the key is placed in taskname+2, which may be examined by user instructions (see the topic ".STATIC SCREEN CODING EXAMPLE" in "Section 11. Terminal I/O"). For a 4979 terminal, PF keys PF1 through PF6 return identifying codes of 1 through 6. Since only the ENTER key and the six PF keys present identifying codes, determining what code to check for is a simple matter.

The 4978 keyboard has a great many more interrupting keys than does the 4979, and determining which key is associated with a particular identifying code is, therefore, more difficult. In fact, by using the DEFINE feature, even the normal alphameric data entry keys and cursor positioning keys may be redefined as interrupting keys.

When \$PFMAP is loaded, it displays, in both decimal and hexadecimal form, the identifying code returned by any interrupting key pressed while \$PFMAP is in execution (with the exception of the ENTER key, which ends the utility). Using this utility, an application programmer can easily find out what code is associated with a particular key and, therefore, what to check for in taskname+2.

## **\$FONT**

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$FONT Utility Program."

As already noted in the discussion of the \$TERMUT2 utility, the 4978 Display has both a control store and an image store. The combination of Scan code, Function ID code, Character/local function code, and Interrupt code which is assigned to each key determines what function each key will perform if pressed. This function definition information is contained in the 4978 control store.

One of the functions which may be assigned to a key is that of displayable character (Function ID code = 00). Each key defined as a displayable character key will have an EBCDIC code associated with it (character/local function code in control store), which will be generated when that key is pressed.

The 4978 also has an image store, containing the bit patterns which, when interpreted by the hardware, result in the display of characters on the screen. Each character bit pattern is associated with an EBCDIC code. When, for example, the "1" key is pressed on the keyboard and a "1" appears on the screen, it is only because of the following circumstances:

1. In the control store, the function ID code for that key position has been defined as 00, assigning that key as a displayable character, and the character/local function code is defined as the EBCDIC character code F1.
2. In the image store, the bit pattern associated with EBCDIC F1 will, when interpreted by the hardware, result in the display of the figure we recognize as the arabic numeral 1.

In the discussion of \$TERMUT2, an example of the CHANGE KEY DEFINITION function was given, whereby control store key definitions could be altered. If, for example, the character/local function code for the 1 key were changed from EBCDIC F1 to EBCDIC C2, pressing the 1 key would result in the display of the alpha character B, because the image store bit pattern associated with C2 is B.

Similarly, the bit patterns in the image store may also be changed to alter the appearance of the characters displayed, or if desired, to create entirely new characters. \$FONT is the utility program used to manipulate image store bit patterns.

4978 image stores, like control stores, may be loaded from disk/diskette data sets. An image is 2K bytes in size, requiring a data set 8 records in length.

```
> [SL SDISKUT1]
SDISKUT1 28P,09:23:01, LP= 7800
USING VOLUME EDX002
COMMAND (?): [AL MYIMAGE 8]
DEFAULT TYPE = DATA - OK? [Y]
MYIMAGE CREATED
COMMAND (?): [END]
SDISKUT1 ENDED AT 09:23:48
```

Figure 14-30. Allocate image store data set

Allocation of a user image store data set is not a prerequisite to using \$FONT. The system-supplied image store data set \$4978ISO can be used with \$FONT. \$4978ISO is, however, automatically loaded to every 4978 supported by the supervisor at IPL, and modifications made will be reflected in all the displays, which may not be desired.

When \$FONT is loaded, the name of an image store data set must be supplied. If not supplied as advance input, as in Figure 14-31, the operator will be prompted to enter it.

```
> [SL $FONT MYIMAGE]
$FONT 22P,09:27:03, LP= 7800
COMMAND(?): [?]
DISP -- DISPLAY TABLE
EDIT -- ENTER EDIT MODE
SAVE -- SAVE TABLE
PUT -- LOAD TABLE INTO DEVICE
GET -- READ TABLE FROM DEVICE
END -- END PROGRAM
COMMAND(?): DISP
```

Figure 14-31. \$FONT commands

The DISP command will display all 256 EBCDIC codes on the screen, along with the characters that are generated for each code by the associated bit patterns in the image store. If DISP were entered at this point, \$FONT would display the image store in the image store data set MYIMAGE. Since MYIMAGE was just allocated, and does not yet contain an image table, a meaningless display would result. To acquire an image table to work with, a GET command may be entered to read an image table from a 4978.

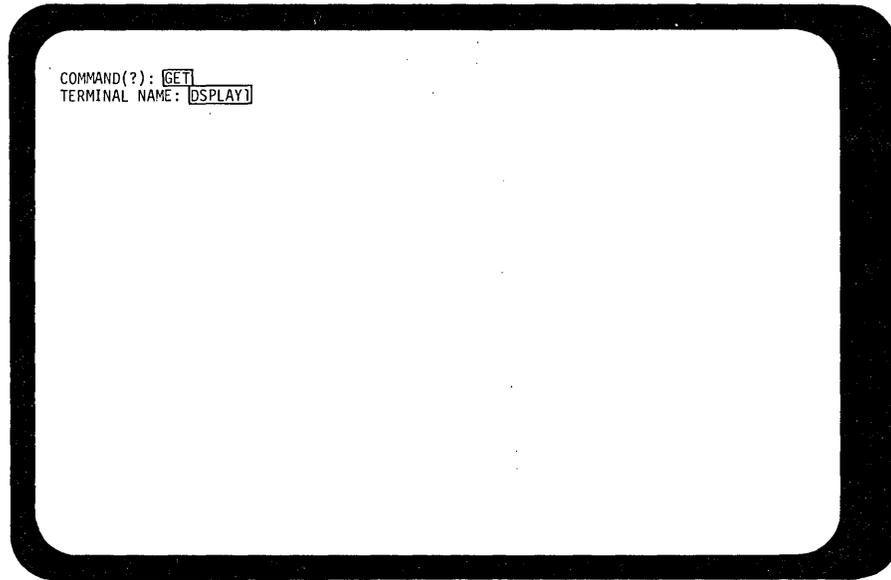


Figure 14-32. GET command

The utility now has an image table, and DISP can be used to display that table.

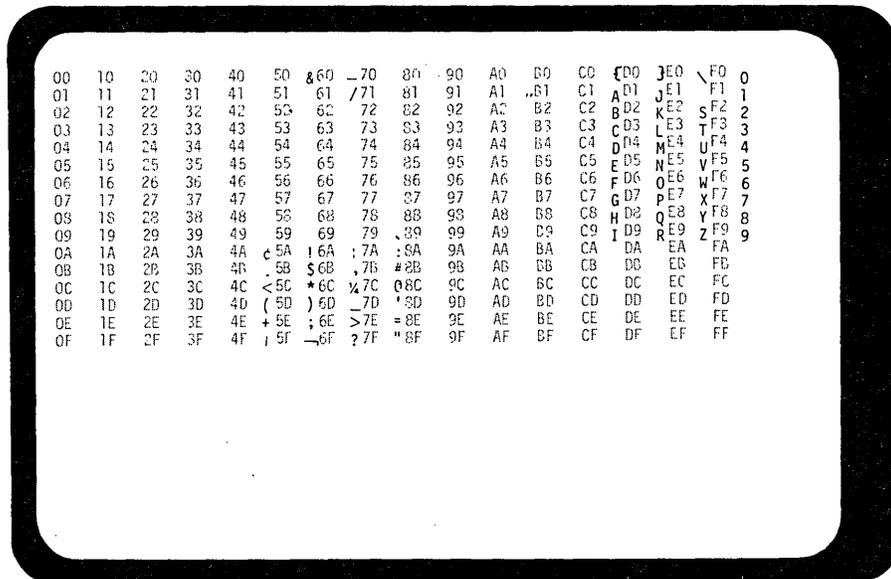


Figure 14-33. Image table

The characters are displayed to the right of the EBCDIC codes with which they are associated. To illustrate how the bit patterns which generate these characters may be altered, assume that the operator wishes to alter the appearance of the character "T" by extending the ends of the crossbar at the top of the character downward. In Figure 14-33, the character "T" is associated with EBCDIC code "E3." To modify or create a character, EDIT mode must be entered. Display mode is ended by pressing the ENTER key. The COMMAND (?): prompt will again appear, whereupon the operator can enter the EDIT command, which will cause the screen in Figure 14-34 to appear.

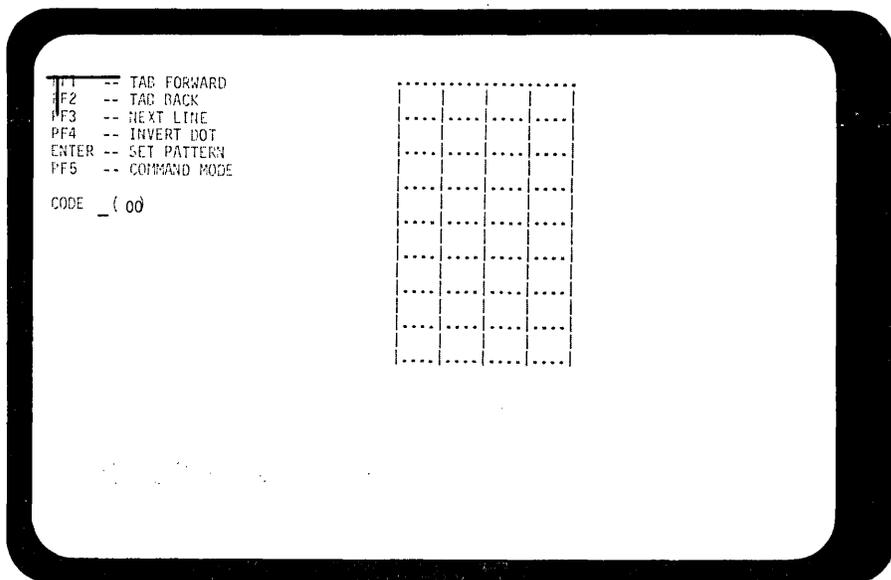


Figure 14-34. EDIT mode (1)

When EDIT mode is first entered, the cursor will be positioned just to the right of the CODE prompt on the bottom left of the screen. The utility is waiting for the operator to enter a character in the present cursor position, or to move the cursor to the right and enter an EBCDIC code between the parentheses. Assume the operator enters the character "T", and presses the ENTER key.

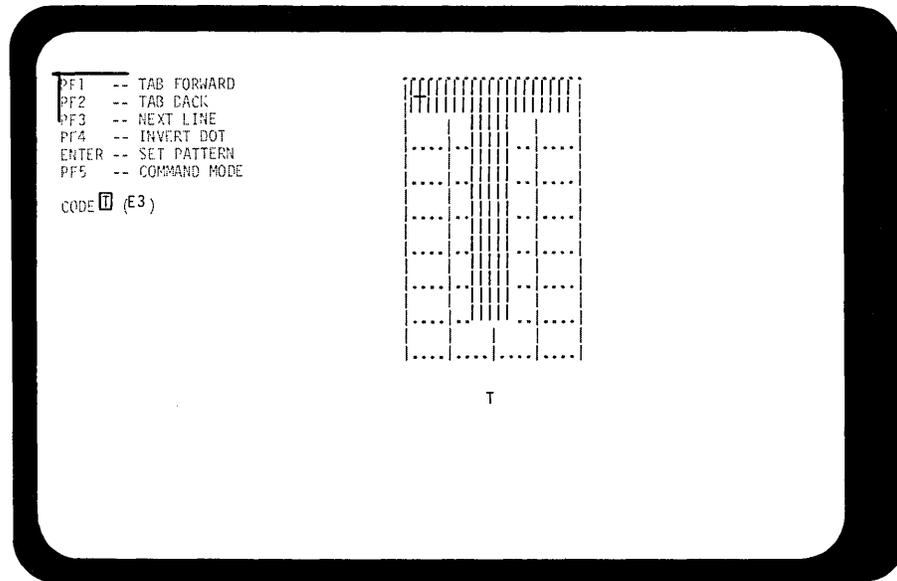


Figure 14-35. EDIT mode (2)

\$FONT fetches the bit pattern for the entered character, displays the character image in the image grid at the right, and places the cursor in the top left-hand square of the 4 by 8 character image grid. The EBCDIC code for the character entered, "E3", is placed in the parentheses to the right of the CODE prompt. The character entered is also displayed below the character image grid (4978 only – not 4979).

The cursor is moved about within the character image grid by the special PF key functions described on the screen. For example, pressing PF1 one time will move the cursor forward (left-to-right, and top-to-bottom) across the grid one position, as shown in Figure 14-36.

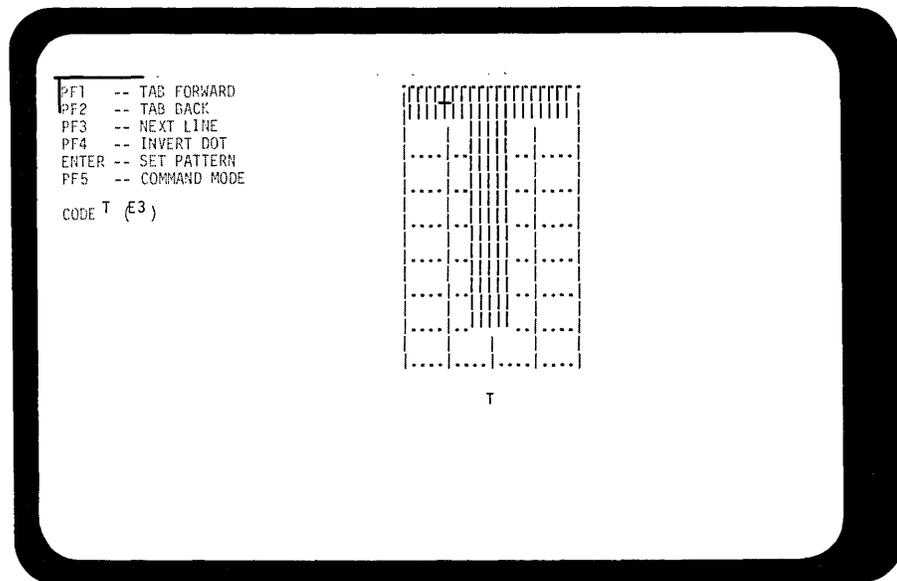


Figure 14-36. EDIT mode (3)

PF2 will move the cursor backwards within the grid (right-to-left and bottom-to-top). PF3 will move the cursor from its present line down to the next line, and position it in the leftmost square of the new line. Figure 14-37 shows the screen after PF3, the "next line" key, has been pressed once.

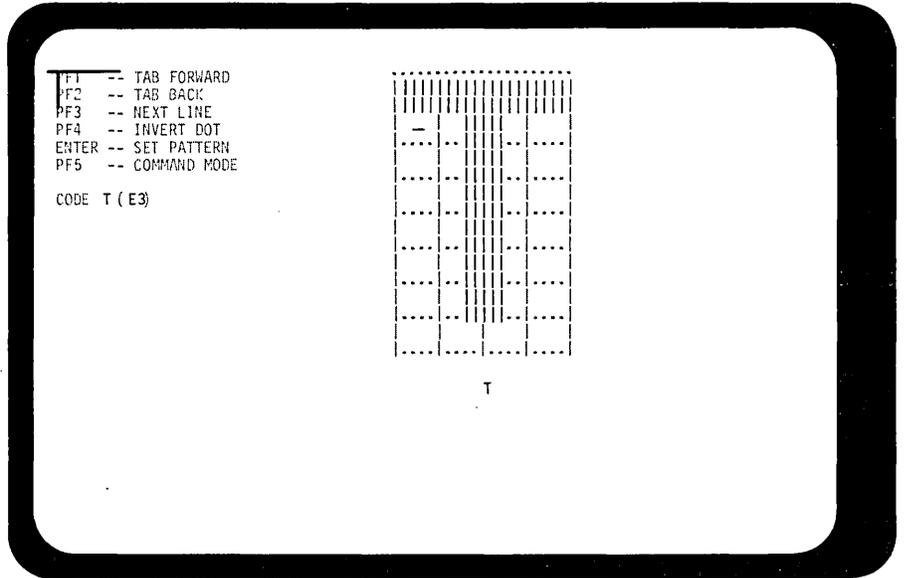


Figure 14-37. EDIT mode (4)

At the outset of this exercise, the stated objective was to alter the appearance of the character "T" by extending ends of the top crossbar downwards. The cursor is now in a position to do that. By hitting PF4, the dot pattern in the first square of the second line is inverted, which extends the left end of the crossbar downwards.

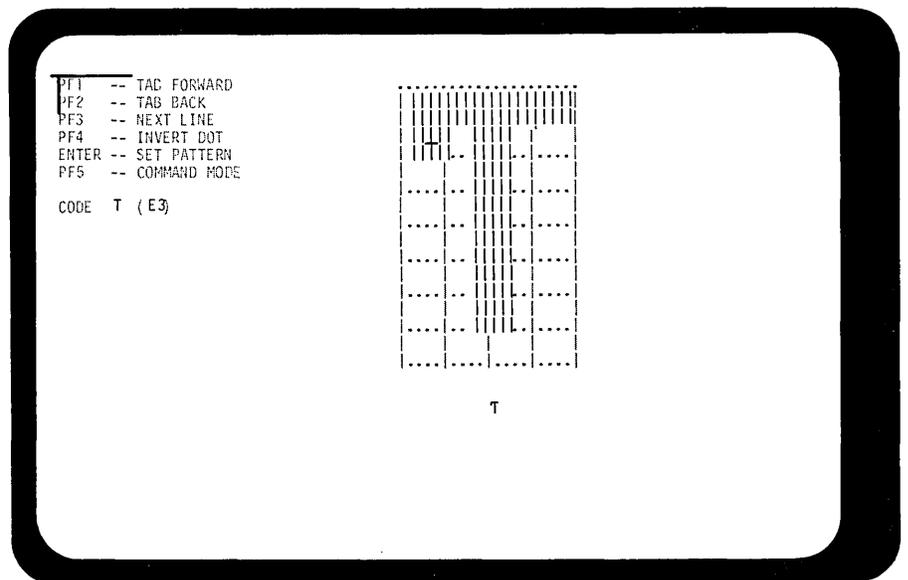


Figure 14-38. EDIT mode (5)

Notice that as modifications are made within the grid, they are reflected in the actual-size character below the grid.

The cursor is now moved forward by pressing PF1 repeatedly until it is at the other side of the grid, as shown in Figure 14-39.

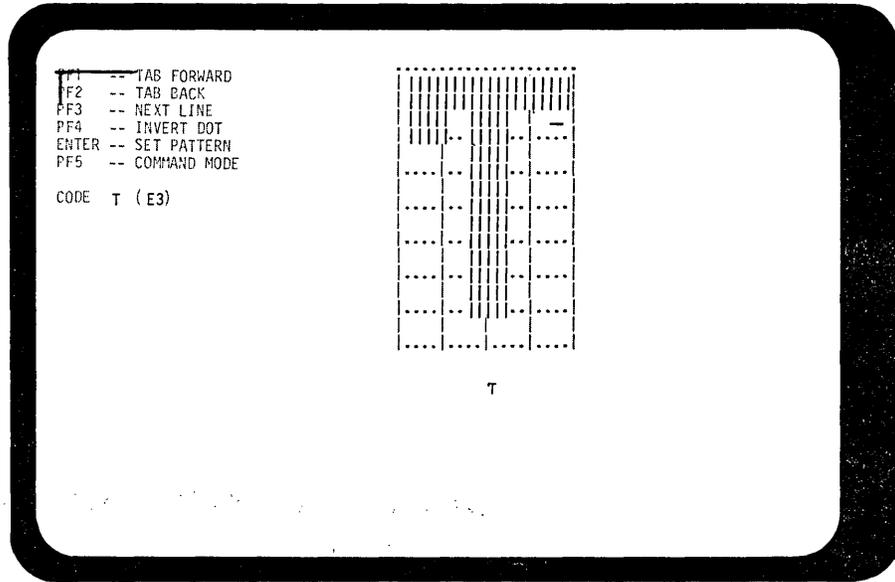


Figure 14-39. EDIT mode (6)

By pressing PF4 again, the other end of the crossbar is extended.

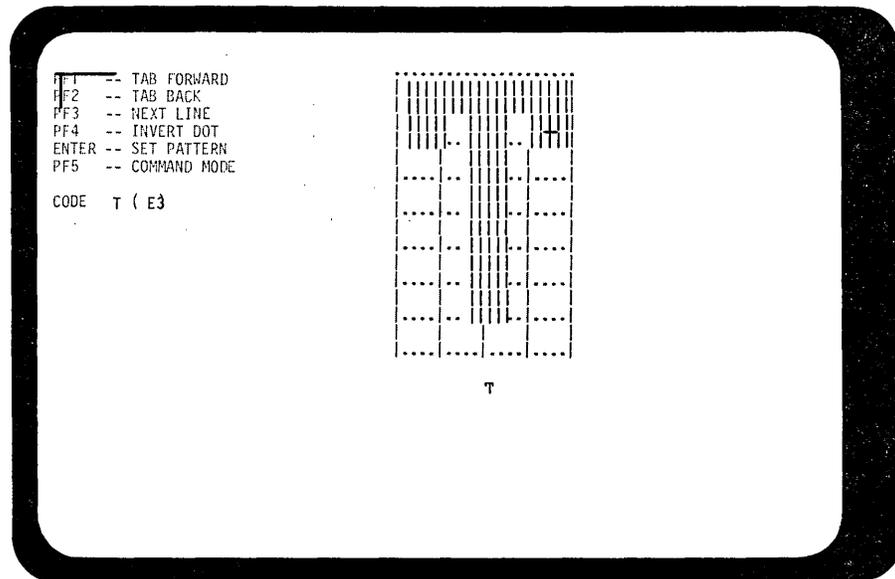


Figure 14-40. EDIT mode (7)

The intended modification is now complete. Pressing the ENTER key results in the screen in Figure 14-41.

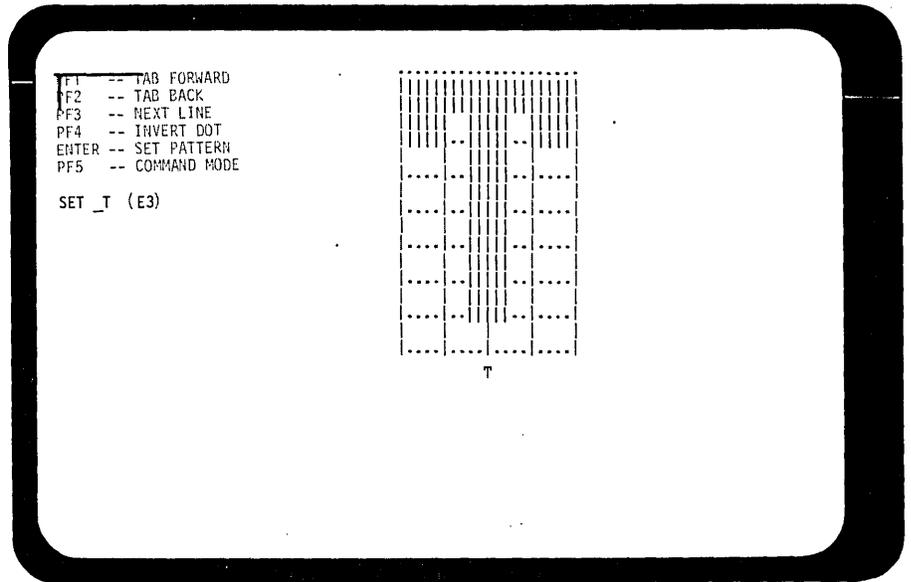


Figure 14-41. EDIT mode (8)

At this point, the operator can "set" the character just composed into the image table. If ENTER is pressed, the modified T will replace the normal T. The operator also has the option of associating the modified T with a different key or EBCDIC code. If, for example, the operator typed an A on top of the T next to the SET prompt, and then pressed ENTER, the modified T would replace the character A. If the operator moved the cursor to the right, and overtyped the E3 within the parentheses with, for example, the EBCDIC code for 0, F0, the modified T image would replace that for 0.

Assume the operator presses ENTER without altering the character or the EBCDIC code, resulting in the screen in Figure 14-42.

```
PF1 -- TAB FORWARD
PF2 -- TAB BACK
PF3 -- NEXT LINE
PF4 -- INVERT DOT
ENTER -- SET PATTERN
PF5 -- COMMAND MODE
CODE T (E3)
```

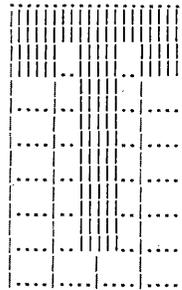


Figure 14-42. EDIT mode (9)

The character has been set, the CODE prompt is again displayed, and the program is waiting for another character or EBCDIC code to be entered. The operator presses PF5, exiting EDIT mode, and reentering command mode.

If you want to be able to load a modified image store to a 4978 at a future time, it must be stored on disk/diskette. The operator therefore enters the SAVE command.

```
COMMAND(?): [ ]
DISP -- DISPLAY TABLE
EDIT -- ENTER EDIT MODE
SAVE -- SAVE TABLE
PUT -- LOAD TABLE INTO DEVICE
GET -- READ TABLE FROM DEVICE
END -- END PROGRAM
```

```
COMMAND(?): SAVE
```

```
COMMAND(?):
```

Figure 14-43. SAVE command

Notice that no data set name is asked for when the SAVE command is entered. The table will always be saved in the data set specified when the utility was loaded; in this case, MYIMAGE.

The only commands not yet exercised are PUT and END. Assuming that this utility session is being conducted on a 4978 named DSPLAY1, a PUT to that device name will load this device with the altered image store, replacing normal "T" characters with new.

```
COMMAND(?): [ ]
DISP -- DISPLAY TABLE
EDIT -- ENTER EDIT MODE
SAVE -- SAVE TABLE
PUT -- LOAD TABLE INTO DEVICE
GET -- READ TABLE FROM DEVICE
END -- END PROGRAM

COMMAND(?): [SAVE]

COMMAND(?): [PUT]
TERMINAL NAME: [DSPLAY1]

COMMAND(?):
```

Figure 14-44. PUT command

This 4978 will continue to display the modified "T" until such time as its image store is again loaded with a different image table (\$TERMUT2, \$FONT, or IPL).

Since the \$FONT utility employs a static screen, this utility can only be used by 4978s or 4979s.

```
COMMAND(?): [?]

DISP -- DISPLAY TABLE
EDIT -- ENTER EDIT MODE
SAVE -- SAVE TABLE
PUT -- LOAD TABLE INTO DEVICE
GET -- READ TABLE FROM DEVICE
END -- END PROGRAM

COMMAND(?): [SAVE]

COMMAND(?): [PUT]
TERMINAL_NAME: [DISPLAY]

COMMAND(?): [END]

$FONT ENDED AT 09:57:44
```

Figure 14-45. END

## MISCELLANEOUS UTILITIES

### \$IMAGE

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$IMAGE Utility Program."

\$IMAGE is used to create formatted screen images for use with terminals that support static screen functions. The images (formatted screens) are stored in disk or diskette data sets for later retrieval by application programs. Stored images may also be retrieved by \$IMAGE for modification/maintenance.

In "Section 16. Program Preparation Using \$EDXASM", the application program used as a program preparation example is the same program used in "Section 11. Terminal I/O" under the topic "STATIC SCREEN CODING EXAMPLE" (see Figure 11-43). In Section 16, the program is modified to retrieve a stored screen image, rather than formatting the screen by executing instructions within the program. The following is a \$IMAGE utility session in which the image that will be used by the modified program is created and stored.

A formatted screen created by \$IMAGE is stored in a disk or diskette data set that must first be allocated by the user. The formatting information and text are stored in a special packed format to conserve space. A logical screen may be of any size from one character position up to an entire physical screen, and therefore the amount of space on disk or diskette required to store a given screen image will vary. For most logical screens, a data set two records in length will be adequate.

The screen image that will be created in this utility session is shown in Figure 14-46 (same as that shown in Figure 11-31). Since it encompasses an entire physical screen and contains several lines of text, a data set three records in length will be required to store it.

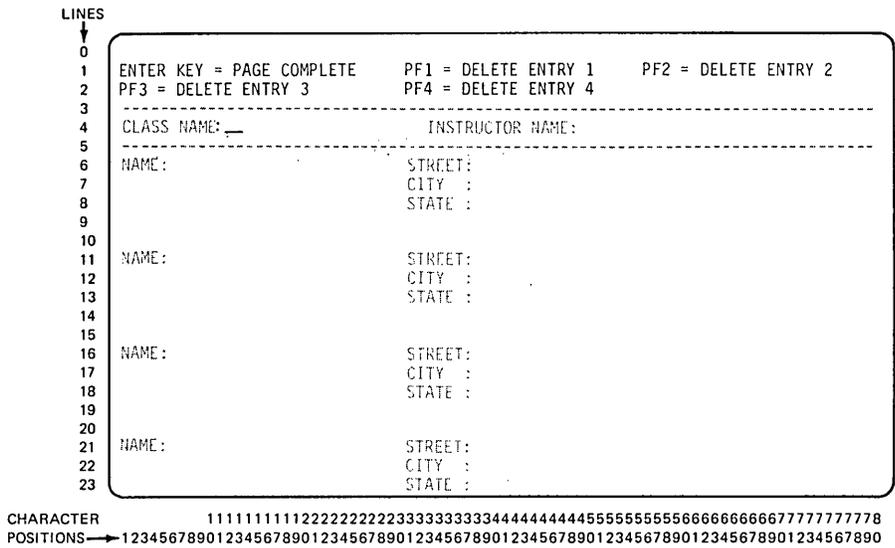


Figure 14-46. Screen image

Before beginning the \$IMAGE utility session, a data set 3 records long, named VIDEO1 is created using \$DISKUT1.

```

> $L $DISKUT1
$DISKUT1      37P,00:32:06, LP= 5F00

USING VOLUME EDX002

COMMAND (?): AL VIDEO1 3
DEFAULT TYPE = DATA - OK? YES
VIDEO1 CREATED

COMMAND (?): END

$DISKUT1 ENDED AT 00:32:33

```

Figure 14-47. Allocate image data set

Now the \$IMAGE utility can be loaded, and the utility session began. Entering a "?" in response to the COMMAND (?): prompt results in a list of the \$IMAGE commands.

```
> $L $IMAGE
$IMAGE          76P, LP= 8D00

COMMAND(?): ?

ATTR - DEFINE ATTRIBUTE CHARACTERS
DIMS - DEFINE SCREEN DIMENSIONS
EDIT - EDIT SCREEN FORMAT
FTAB - DISPLAY FIELD TABLE
HTAB - SET (HORIZONTAL) TABS FOR EDIT
KEYS - DISPLAY USE OF PF KEYS
NULL - DEFINE NULL CHARACTER
PRNT - PRINT SCREEN IMAGE AND TABLES
SAVE - SAVE SCREEN FORMAT(S) IN DATASET
VTAB - SET (VERTICAL) TABS FOR EDIT
END - END PROGRAM

COMMAND(?): ATTR

ATTRIBUTE CHARACTERS WITH MDT OFF:

LOW INTENSITY (NOW IS ' ') = %
HIGH INTENSITY (NOW IS ' ') = *
BLINKING      (NOW IS ' ') = $
NONDISPLAY    (NOW IS ' ') = #

DO YOU WISH TO DEFINE
ATTRIBUTE CHARACTERS WITH MDT ON? N

COMMAND(?): DIMS 24 80
COMMAND(?): HTAB 31
COMMAND(?): NULL /
COMMAND(?): EDIT
```

Figure 14-48. \$IMAGE commands

The ATTR command allows you to define the characters that define the attribute bytes for the 3101 M2 screen. When creating a screen, the display mode of a field (protected or unprotected) can be set by prefixing the field with an attribute byte. In the example placing a \$ in front of a field will cause that field to blink when the screen is displayed. Attribute bytes are specified when defining protected or unprotected null fields.

The DIMS command allows you to define the dimensions of the logical screen you are creating. The example shows a logical screen of 24 lines and 80 characters specified, which is equal to the entire physical screen.

HTAB is the horizontal tab settings you wish to have in effect while you are creating the screen. If not entered, HTAB defaults to 10, 20, 30 etc, through 80. The example defines a single HTAB setting of 31.

VTAB defines vertical tabs. The default is one vertical line for each vertical tab key depression. Since VTAB is not entered in this example, one-line vertical tabs will be in effect.

The NULL command allows you to define the null character. When in EDIT mode, a null character is entered in each character position you want to display unprotected data in, or in which operator-entered data is to be accepted, when the completed screen is used by an application program.

The KEYS command lists the functions of PF1, PF2, and PF3 (functions valid when EDIT mode is entered).

PF1—define protected fields  
PF2—define data fields (unprotected)  
PF3—return to COMMAND mode

Figure 14-49. KEYS

All of the commands listed in Figure 14-48 may only be entered in the COMMAND mode. The last command entered (Figure 14-48) is EDIT, which places the \$IMAGE utility in EDIT mode. If an existing screen image were to be edited, the data set name and volume of that image would be entered with the EDIT command. Since this session is creating a new screen, EDIT is entered without reference to a data set.

When EDIT mode is entered, PF1, PF2, and PF3 have the functions listed in Figure 14-49. Before pressing any of the PF keys, the screen is entirely blank, and the cursor is in the lower left corner.

The logical screen being created in this example contains both protected and unprotected data. The operator prompts on lines 1 and 2 are unprotected, and the rest of the prompts are protected (see Figure 14-46). When the completed screen is displayed, the unprotected areas will appear brighter than those that are protected, highlighting the prompts at the top of the image.

When both protected and unprotected text is to appear on a screen created by \$IMAGE, the protected data must be entered first. Therefore PF1 is depressed, signalling to the utility that protected fields are to be defined. The cursor now moves to the first available character position, which is line 0, space 0, in this example.

As soon as either PF1 or PF2 is pressed, after entering EDIT mode, the function of PF1 and PF2 is redefined. PF1 is now used as the horizontal tab key, and PF2 as the vertical tab key. Since no text appears on line 0, the vertical tab key PF2 is pressed, moving the cursor down to the first position of line 1.

When defining the protected areas of a screen image, all characters entered, other than the null character, will be protected data. The operator prompts on lines 1 and 2 are supposed to be unprotected. Therefore, the actual text of the prompts cannot be entered until the data definition portion of this utility session, after all protected fields have been defined. However, since these areas of the screen will contain unprotected text, null fields must be established, so that when the unprotected data definition is done, the text entered will be accepted. Figure 14-50 shows the screen after the null characters for the unprotected operator prompts at the top of the screen have been entered.

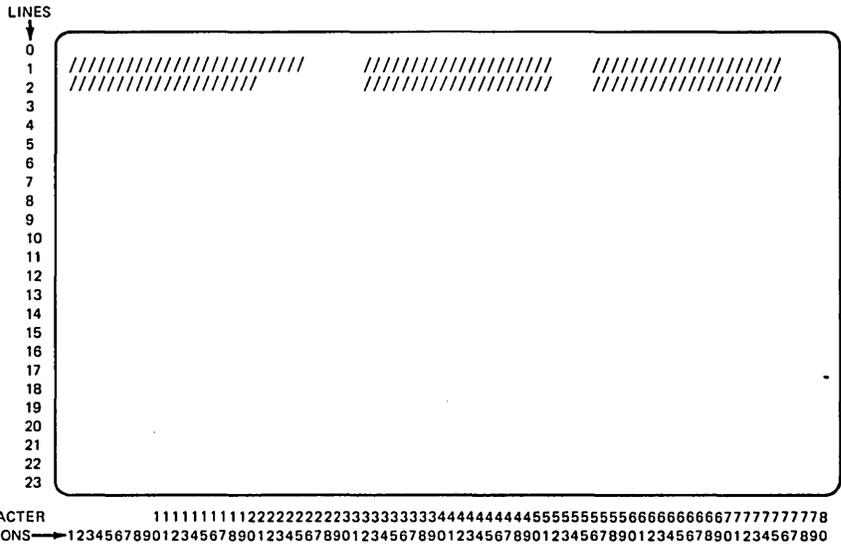


Figure 14-50. NULL entries

Now the rest of the screen can be formatted. All areas of the screen not containing null characters will be protected when the screen is completed. Note that any field meant to receive operator input when the screen is used must be defined using the null character.

Figure 14-51 is the screen after all protected data has been defined.

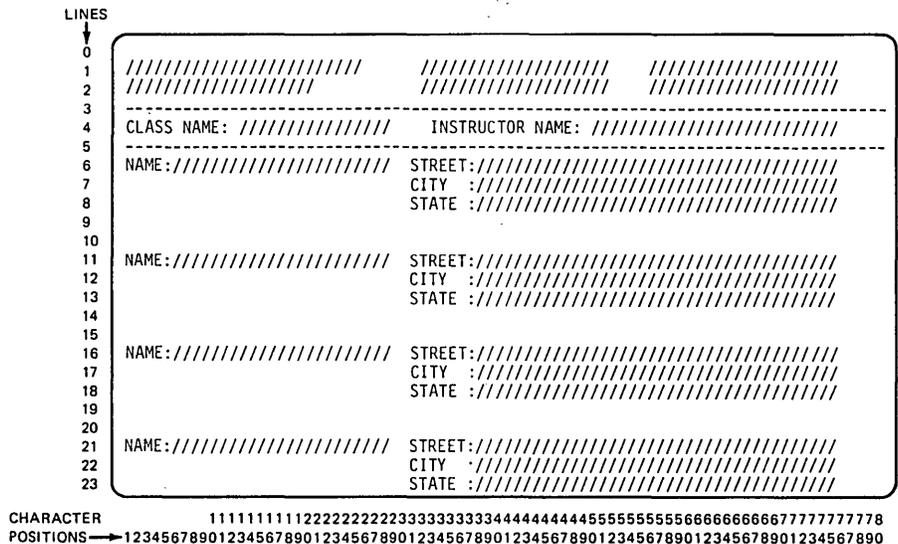


Figure 14-51. Protected entries

Pressing the ENTER key takes the utility out of protected field definition, back to EDIT mode (the situation as it was before a define protected field or define data field decision was made). PF1 and PF2 again have the meanings printed out by the KEYS command (Figure 14-49). The ENTER key also causes the screen, as defined up to this point, to be displayed as pictured in Figure 14-52.

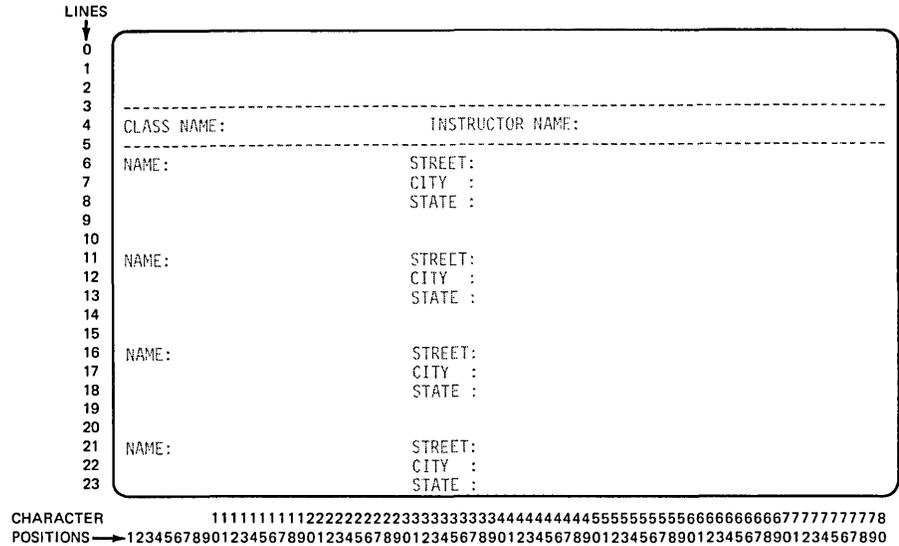


Figure 14-52. Partially completed image

If the desired screen image were now complete, PF3 would be pressed to get back into COMMAND mode, so that it could be saved. In this example, however, there is still unprotected data to be defined, so PF2 is pressed. PF2 brings back the same screen image as in Figure 14-51, with the unprotected fields defined as null characters.

The unprotected null fields in the operator prompt area at the top of the screen are now filled in. The other null fields are input fields that will be used when the screen is used by an application program, so are left undisturbed during screen creation.

After all unprotected text is defined, the screen looks like that shown in Figure 14-53.

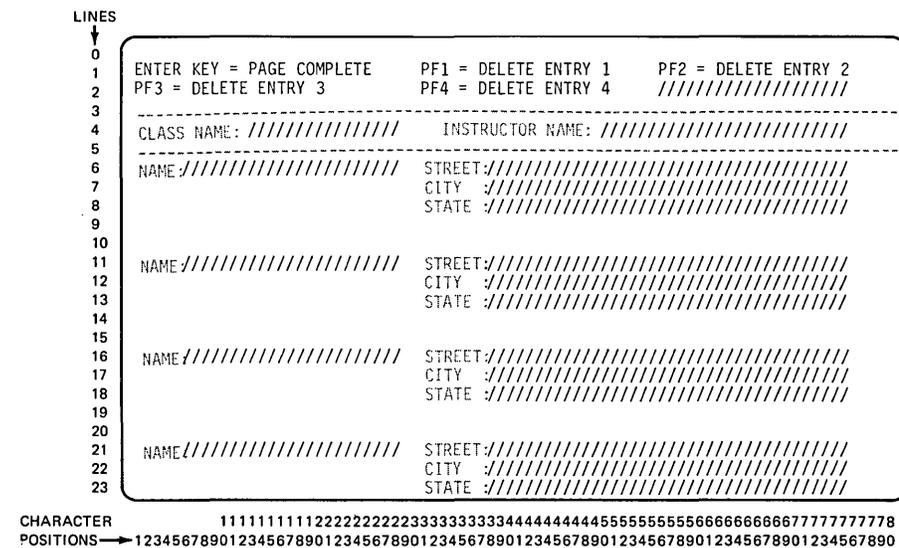


Figure 14-53. Complete image

When ENTER is pressed, the completed screen is displayed (Figure 14-46). Any desired changes can be made by again pressing PF1, for protected fields, or PF2, for unprotected ones. Assuming that the image is correct, the operator will press PF3 to return to COMMAND mode.

PF3 will blank the screen, and prompt for a command entry.

```
COMMAND(?): 
```

```
COMMAND(?): 
```

Figure 14-54. Save image

The operator enters the SAVE command, followed by the name of the data set that was allocated for this purpose. The \$IMAGE utility session is then ended.

## \$IOTEST

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$IOTEST Utility Program."

\$IOTEST is an exerciser program for the digital and analog sensor I/O features of the Series/1. The operator is prompted for various operating parameters, and the \$IOTEST utility then repetitively executes the requested exercising operations. See the reading assignment for examples of the use of this program.

The LD function will list all the hardware devices, and their hexadecimal addresses, for the Series/1 on which \$IOTEST is running. The LS command will list the hardware devices and addresses supported by the supervisor currently in operation. These two functions are particularly useful during system generation. By comparing the output produced by the LD and LS commands, the operator can easily spot devices attached but not supported, supported and not attached, or attached and supported, but assigned the wrong address. Use of these \$IOTEST options is illustrated in "Section 15. System Installation."

## \$PREFIND

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$PREFIND Utility Program."

If a program uses data sets or overlay programs (DS= and PGMS= keyword parameters in PROGRAM statement), the assembly process creates control blocks in the program header for each data set and overlay program specified. Space is reserved in these control blocks for the physical disk/diskette addresses of the data sets and overlay programs defined.

After completion of the program preparation process (\$LINK if required, and then \$UPDATE), the executable load module may be loaded to storage. The system program that performs the load operation is \$LOADER, and part of that operation includes filling in the actual physical addresses of data sets and overlay programs in the control blocks of the program header. When a large number of data sets and/or overlay programs are defined, this can be a time-consuming process, as \$LOADER must search a volume directory for each data set/program used.

To illustrate, the example program in Figure 14-55 has been assembled and formatted (\$UPDATE), and stored on disk under the name MAIN.

```
00001 PFNDEXMP PROGRAM   START,DS=(DMY1,DMY2,DMY3,DMY4,DMY5),PGMS=(OVLY1,   C
00002                   OVLY2,OVLY3,OVLY4,OVLY5)
00003 START      PROGSTOP
00004           ENDPROG
00005           END
```

Figure 14-55. Source for MAIN

This program has five data sets and five overlay programs defined. Since any data sets or overlay programs used by a program must exist at the time the program is loaded, five 1-record data sets, DMY1 through DMY5 have been created using the AL function of \$DISKUT1.

Figure 14-56 is the source used to create the overlay programs.

```
00010 OVLYPROG PROGRAM   START
00020 START      PROGSTOP
00030           ENDPROG
00040           END
```

Figure 14-56. Overlay source

The object module produced by assembly of this source has been successively processed by \$UPDATE five times, each time providing a different load module name, OVLY1 through OVLY5.

Figure 14-57 is a load request for program MAIN.

```

                                LOAD REQUEST ENTERED
                                AT 01:00:20
> $L MAIN
MAIN                               4P,01:00:43, LP= 7800

MAIN                               ENDED AT 01:00:43

```

Figure 14-57. Load request

The amount of time elapsed, from the time the ENTER key is pressed to enter the \$L MAIN command to the time the load message is returned, is 23 seconds by stopwatch. The majority of this time was taken in looking up the data set and overlay program locations for the control blocks in the program header.

\$PREFIND allows data set and overlay programs to be located prior to program load time, and written directly into the program header on disk/diskette. When the program is loaded, the information required is already present, and load time is therefore reduced. In Figure 14-58 the example program MAIN is processed by \$PREFIND.

```

> $L $PREFIND
$PREFIND 26P,00:57:40, LP= 7800
COMMAND (?): ?
          PF  PRELOCATE DATA SETS AND OVERLAYS
          DE  DELETE PRE-FOUND STATUS
          EN  END THE PROGRAM
COMMAND (?): PF
PGM(NAME,VOLUME): MAIN
ENTER DATA SET NUMBERS: D=(1,2,3,4,5)
ENTER OVERLAY PGM. NUMBERS: P=(ALL)

COMMAND COMPLETED
COMMAND (?): END

```

Figure 14-58. \$PREFIND operation

In Figure 14-58, the "D=" entry could have been "ALL", just as it is for the "P=" entry, with the same effect. If you don't want all data sets or programs prelocated, you can selectively pre-find only those you wish, by entering the desired positional reference numbers and leaving out those you want the loader to find at load time.

After the operation shown in Figure 14-58, load time for program MAIN is under three seconds.

Once a program has been processed by \$PREFIND and "pre-found" status is established, the system makes no further checks to verify the validity of the data set/overlay program addresses in the program header. The AL and DE functions of \$DISKUT1, or operations involving use of \$COPYUT1, \$UPDATE, and \$COMPRES can alter program/data set locations, and therefore invalidate a program's pre-found addresses. This could result in reading the wrong data, writing over important data, etc. \$PREFIND is therefore not appropriate for use in test/development environments, and even in "stable" application environments should only be used with care.

## PROGRAM PREPARATION UTILITIES

### \$EDIT1N

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$EDIT1 and \$EDIT1N Utility Programs."

The \$EDIT1N text editing utility is used to create and edit source programs and other text data records such as the procedure files used with \$JOBUTIL, or the control record files for \$LINK. \$EDIT1N (and also \$EDIT1 and \$FSEDIT) uses a data member as an edit work area. This work file must be preallocated by the user (\$DISKUT1), and must be of sufficient size to contain the largest source program anticipated. The required size can be calculated as follows: number of text lines (n) divided by 30 times 11 plus 1 ( $n/30 \times 11 + 1$ ). The four primary text editor commands are:

1. READ — get the contents of a data set on a specified logical volume and store it in the work area data set.
2. LIST — list the contents of the work area on the system printer (for the starter system on the matrix printer).
3. END — terminate the text editor.
4. EDIT — go into edit mode allowing the user to use any of the edit subcommands.

Figure 14-59 is an example of a text edit session, demonstrating several of the EDIT mode subcommands. \$EDIT1N is also used to edit the system configuration statements and link editor INCLUDE statements during system generation. See the topic "USER SYSTEM GENERATION" in "Section 15. System Installation."

```

EXAMPLE: > $L $EDIT1N ← 1
          DS1(NAME,VOLUME): EDITWORK ← 2
          $EDIT1N      47P,00:08:01, LP= 5100
          READY
          EDIT ← 3

          EDIT
          DE 10 250 ← 4
          TOP OF DATASET
          INPUT ← 5
          INPUT
          00010 %PRINT%NOGEN
          00020 PGM1%PROGRAM4%START,100
          00030 START%PRINTEXT%TXT1,SKIP=2
          00040 %ATTACH%TASK1
          00050 %WAIT%E1,RESET
          00060 %PROGSTOP
          00070 TXT1%TEXT%'PROGRAM STARTED' ← 6
          00080 TXT"%TEXT%'TASK1 RUNNING'
          00090 TASK1%TASK%GO,EVENT=E1
          00100 TASK1%TASK%GO,EVENT=E1
          00110 GO%PRINTEXT%TXT2
          00120 %ENDPROG
          00130 %END
          00140 ← 7

          EDIT
          CHANGE 80 /T"/T2/ ← 8
          DELETE 100 ← 9
          INPUT 115 ← 10
          INPUT
          00115 %ENDTASK
          INPUT TERMINATED
  
```

Figure 14-59. \$EDIT1N (1 of 2)

```

EDIT
LI ← 11
00010          PRINT      NOGEN
00020 PGM1     PROGRAM4  START,100
00030 START   PRINTEXT  TXT1,SKIP=2
00040          ATTACH    TASK1
00050          WAIT      E1,RESET
00060          PROGSTOP
00070 TXT1    TEXT       'PROGRAM STARTED'
00080 TXT2    TEXT       'TASK1 RUNNING'
00090 TASK1   TASK      GO,EVENT=E1
00110 GO      PRINTEXT  TXT2
00115          ENDTASK
00120          ENDPROG
00130          END
END OF DATA
SAVE ← 12
ENTER VOLUME LABEL: EDX001
ENTER MEMBER NAME: COPY
END AFTER      13

IODA,CTS= 002,047013,049010

READY ← 13
END ← 14

$EDIT1N ENDED AT 00:23:11

```

Figure 14-59. \$EDIT1N (2 of 2)

#### COMMENTS:

- 1** The Text Editor is loaded.
- 2** A preallocated data set to be used as a work area is specified.
- 3** If you were updating a source module you would issue a READ indicating the data set name and volume that contain the file. In this example a new source module is being created, so EDIT mode is invoked without a preceding READ.
- 4** This DELETE removes text lines remaining from a previous editing session (clears the work area) and positions the editor at the beginning (TOP) of the work area.
- 5** To enter source statements you must issue the INPUT subcommand.
- 6** The source statements entered are shown. The % in the text is used as the default TAB character.
- 7** To end the INPUT subcommand depress the ENTER key or carriage return without entering any data.
- 8** An error was made in the original entry on line 80. The slash is the delimiter between the change fields. Any non-numeric (except blank, TAB character or \*) can be used as the delimiter. Here T2 replaces "T" in line 80.

- 9 Another error was made in the original input. Line 90 and 100 are the same. Line 100 is deleted.
- 10 The user forgot to end the task with an ENDTASK instruction. It is now entered as line 115.
- 11 Using the EDIT subcommand LIST, the contents of the work area are listed on the terminal. A LIST subcommand issued when not in EDIT mode will list the work area on the system printer.
- 12 The data in the work area is now saved in a preallocated user data set. The SAVE operation translates the source statements from the text editor format, in which they exist in the work area, into the normal source statement format which can be accepted by the assembler. The save is not destructive; the data is retained in the work area.
- 13 When the SAVE is complete, EDIT mode terminates.
- 14 To terminate the text editor, key in END.

## \$UPDATE

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$UPDATE and \$UPDATEN Utility Programs."

\$UPDATE is the utility used to format object modules into relocatable load modules, which can be loaded to storage and executed.

COMMAND: RP

FUNCTION: Read a program and convert it to a relocatable load module.

```
EXAMPLE: > $L $UPDATE
          $UPDATE      33P,00:00:20, LP= 5100

          THE DEFINED INPUT VOLUME IS EDX002, OK?  Y
          THE DEFINED OUTPUT VOLUME IS EDX002, OK?  Y

          COMMAND (?):  RP

          OBJECT MODULE NAME:  DEMO

          OUTPUT PGM NAME:  FMT
          FMT REPLACE?  Y
          FMT STORED
```

Figure 14-60. \$UPDATE

COMMENTS: This example shows the formatting of an object module, DEMO. The executable output program, FMT, is stored. If a program member with the same name exists, you will be asked if it is to be replaced. If it does not exist, the utility will allocate the space for the executable program. The program, FMT, in the example can now be loaded by the \$L operator command or by a LOAD instruction in a program.

## **\$FSEDIT**

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$FSEDIT Utility Program."

This utility provides full-screen text editing capability for the Event Driven Executive. \$FSEDIT operates the terminal as a static screen device, and therefore must be run from a terminal with static-screen capability (4978/4979/3101 M2).

*Data Set Requirements.* \$FSEDIT requires a preallocated work data set for use as a text edit work area. Text data (source statements) within this work data set are in a special text editor format, identical to that used by the \$EDIT1N text editor; data within a text edit work data set may be edited by either \$EDIT1N or \$FSEDIT.

At the conclusion of a text edit utility session, it is important to save the contents of the edit work data set in a source data set on disk or diskette (automatic translation from text editor format to source statement format is performed).

\$FSEDIT is loaded using \$L operator command (the operator must provide the name of a text edit data set when the load request is entered). The operator will be prompted for the names of input/output source data sets during the utility session, at the time a READ or WRITE option is selected

### *\$FSEDIT Primary Options*

When \$FSEDIT is first loaded, the screen shown in Figure 14-61 will be displayed, with the cursor positioned just to the right of the SELECT OPTION arrow. An option is selected by entering a number corresponding to the desired option, and pressing the ENTER key.

```
----- $FSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==> _
1 BROWSE - DISPLAY DATASET
2 EDIT - CREATE OR CHANGE DATASET
3 READ - READ DATASET FROM HOST/NATIVE (H/N)
4 WRITE - WRITE DATASET TO HOST/NATIVE (H/N)
5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
6 LIST - PRINT DATASET ON SYSTEM PRINTER
7 MERGE - MERGE DATA FROM A SOURCE DATASET
8 END - TERMINATE $FSEDIT
9 HELP - DISPLAY TUTORIAL
```

Figure 14-61. \$FSEDIT (1)

Option 5: SUBMIT is used to submit a job to a host program preparation system, and will therefore not be discussed in this section. The rest of the options will be illustrated in the order in which they would normally be required, not in the numerical sequence in which they appear in Figure 14-61.

### *Creating A Source Statement File*

When the Primary Option Menu is displayed (Figure 14-61), entering a 2 places the utility in EDIT mode. EDIT mode is used to modify an existing source data set, or to create a new one. When modifying an existing data set, a READ (option 3) of the file to be modified, into the edit work data set, must first be performed. This will be illustrated later. At this point, assume a new source statement file will be created.

Invoking EDIT mode with an empty edit work data set will result in display of the screen in Figure 14-62. Because the work data set is empty, the editor assumes insertion (creation) of lines is desired, and the INSERT function is therefore active. The five dots to the left of the cursor will contain the statement number of the new line once it has been entered. The cursor is positioned at character position 1 of the insert line.

```
EDIT --- EDITWORK, EDX002      0( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
***** ***** TOP OF DATA *****
.....
*****
```

Figure 14-62. \$FSEDIT (2)

The top line of the screen, from left to right, displays the mode the utility is in (EDIT), the name and volume of the work data set (EDITWORK,EDX002), the number of source statements in the work data set, and in parentheses, the total number of statements the data set will hold.

In Figure 14-63, a line of asterisks and spaces has been entered on the insert line, and the ENTER key pressed. The utility numbers the entered line and sets up for the next insert line.

```
EDIT --- EDITWORK, EDX002      1( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
***** ***** TOP OF DATA *****
00010 [*****]
.....
*****
```

Figure 14-63. \$FSEDIT (3)

Notice that the "number of source statements in work data set" value on the top line has incremented.

Continuing in this manner, with a new insert line readied each time the preceding line has been entered (ENTER key), the 18 comment statements (asterisk in position 1) shown in Figure 14-64 are created. The insert operation is terminated by pressing the ENTER key without entering anything on the new insert line.

```
EDIT --- EDITWORK 00102 151 2701----- COMMENTS 001 072
COMMAND INPUT ==:MENU CL ==>HALF
***** TOP OF DATA *****
00010 *
00020 *
00030 * THIS SET OF COMMENT STATEMENTS DEMONSTRATES THE ABILITY TO CREATE
00040 * A SOURCE FILE, BEGINNING WITH AN EMPTY WORK DATA SET. WHEN
00050 * COMPLETE, THIS SET OF STATEMENTS WILL BE WRITTEN TO THE PRE-
00060 * ALLOCATED DATA SET "MGRDATA" ON VOLUME EDX002. A PORTION OF DATA
00070 * SET "MGRDATA" WILL BE USED LATER TO ILLUSTRATE THE "MERGE"
00080 * PRIMARY OPTION OF $FSEDIT.
00090 *
00100 *
00110 *
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * MERGE DATA
00160 * MERGE DATA
00170 *
00180 *
***** BOTTON OF DATA *****
```

Figure 14-64. \$FSEDIT (4)

The cursor is automatically positioned to the right of the COMMAND INPUT arrow on the second line from the top of the screen. To return to the Primary Option Menu, the command "MENU" is entered, and the ENTER key pressed. This brings back the screen shown in Figure 14-61.

*Option 4: WRITE*

The source statements just created will now be saved as a source data set. The WRITE primary option is selected, and the operator is prompted for the target data set/volume on the bottom half of the screen, as shown in Figure 14-65.

```
-----4N-- $FSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

 1 BROWSE - DISPLAY DATASET
 2 EDIT   - CREATE OR CHANGE DATASET
 3 READ   - READ DATASET FROM HOST/NATIVE (H/N)
 4 WRITE  - WRITE DATASET TO HOST/NATIVE (H/N)
 5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
 6 LIST   - PRINT DATASET ON A SYSTEM PRINTER
 7 MERGE  - MERGE DATA FROM A SOURCE DATASET
 8 END    - TERMINATE $FSEDIT
 9 HELP   - DISPLAY TUTORIAL
-----

ENTER DATASET (NAME,VOLUME): SOURCE, ASMVOL
```

Figure 14-65. \$FSEDIT (5)

After the contents of the work data set have been written, the prompt will be replaced by an ending message indicating how many statements had been written; in this example END AFTER 18. The cursor is returned to the SELECT OPTION input area.

### Option 3: READ

To edit an existing source file, it must first be transferred to the edit work data set. A diskette volume called ASMVOL is mounted, which contains a data set named SOURCE. By entering 3 and responding to the resulting prompts as shown in Figure 14-66, this file is read into the edit work data set.

```
-----3--- $FSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

 1 BROWSE - DISPLAY DATASET
 2 EDIT   - CREATE OR CHANGE DATASET
 3 READ   - READ DATASET FROM HOST/NATIVE
 4 WRITE  - WRITE DATASET TO HOST/NATIVE
 5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
 6 LIST   - PRINT DATASET ON SYSTEM PRINTER
 7 MERGE  - MERGE DATA FROM A SOURCE DATASET
 8 END    - TERMINATE $FSEDIT

-----

READ FROM NATIVE?  YES

ENTER VOLUME LABEL: 
ENTER MEMBER NAME: 
```

Figure 14-66. \$FSEDIT (6)

*Option 6: LIST*

Entering primary option 6 will list the contents of the work data set on the system printer. The data set SOURCE on ASMVOL contains the source file for the program used as an example in "Section 11. Terminal I/O". Listing the contents of the edit work area will produce the same listing as that shown in Figure 11-43, but with statement numbers printed to the left of each statement.

*Option 1: BROWSE*

The BROWSE option is used to examine a source file in the edit work data set, while precluding the possibility of changing it. Paging response will generally be faster in this mode. If option 1 is entered with the work data set containing the file from data set SOURCE, the screen in Figure 14-67 will be displayed. Note again the top line of the screen, indicating the operating mode (BROWSE) and the size of the file being examined (75 statements).

```

BROWSE - EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> PAGE
**** ** TOP OF DATA ****
00010 XPLSTAT PROGRAM  START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
00040 ATTNLIST (END,OUT,$PF,STATIC)
00050 START ENQT IOCB1
00060 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00070 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00080 PRINTTEXT 'THE PROGRAM'
00090 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00100 PRINTTEXT 'BRING UP THE ENTRY SCREEN'
00110 DEQT
00120 CHECK WAIT ATTNECB,RESET
00121 IF (ATTNECB,EQ,1),GOTO,ENDIT
00140 ENTRY ENQT IOCB2
00150 ERASE MODE=SCREEN,TYPE=ALL
00160 TERMCTRL BLANK
00170 PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00180 PRINTTEXT ' PF1 = DELETE ENTRY 1'
00190 PRINTTEXT ' PF2 = DELETE ENTRY 2'
00200 PRINTTEXT 'PF3 = DELETE ENTRY 3' ',SKIP=1
00210 PRINTTEXT 'PF4 = DELETE ENTRY 4'

```

Figure 14-67. \$FSEDIT (7)

This file, as with most source files, is too large to be displayed in its entirety on the screen. In Figure 14-67, only the first 21 of the 75 statements which make up the file are in view.

To allow viewing of all parts of a file, both BROWSE (option 1) and EDIT (option 2) modes have a "scrolling" function, invoked by pressing PF keys. PF3 is used to scroll down in the data set, from top to bottom, and PF2 to scroll up, from bottom to top.

In Figure 14-67, the scroll size is displayed at the extreme right of the second line. In BROWSE mode, the normal scroll size is PAGE; 22 lines of data. In Figure 14-68, PF3 has been pressed, displaying the next 22 lines in the work area (statements 220 through 430).

```

BROWSE - EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>PAGE
00220      PRINTEXT DASHES,PROTECT=YES,LINE=3
00230      PRINTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
00240      PRINTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00250 HDR  PRINTEXT DASHES,PROTECT=YES,LINE=5
00260      MOVE     LINENBR,6
00270      DO       4,TIMES
00280      PRINTEXT 'NAME:',LINE=LINENBR,PROTECT=YES
00290      PRINTEXT 'STREET:',LINE=LINENBR,SPACES=30,PROTECT=YES
00300 A1   ADD     LINENBR,1
00310      PRINTEXT 'CITY :',LINE=LINENBR,SPACES=30,PROTECT=YES
00320 A2   ADD     LINENBR,1
00330      PRINTEXT 'STATE :',LINE=LINENBR,SPACES=30,PROTECT=YES
00340      ADD     LINENBR,3
00350      ENDDO
00360      PRINTEXT LINE=4,SPACES=11
00370      TERMCTRL DISPLAY
00380 WAITONE WAIT   KEY
00390      GOTO   (READ,E1,E2,E3,E4),XEMPLSTAT+2
00400 E1   MOVE   LINENBR,6
00410      GOTO   DELETE
00420 E2   MOVE   LINENBR,11
00430      GOTO   DELETE

```

Figure 14-68. \$FSEDIT (8)

The scroll size may be defined as HALF by moving the cursor to the scroll size area and entering HALF where PAGE now is. HALF indicates half a page, or 11 lines. In Figure 14-69, scroll size has been defined as HALF, and PF3 has been pressed, displaying 11 new lines of data.

```

BROWSE - EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
00330      PRINTEXT 'STATE :',LINE=LINENBR,SPACES=30,PROTECT=YES
00340      ADD     LINENBR,3
00350      ENDDO
00360      PRINTEXT LINE=4,SPACES=11
00370      TERMCTRL DISPLAY
00380 WAITONE WAIT   KEY
00390      GOTO   (READ,E1,E2,E3,E4),XEMPLSTAT+2
00400 E1   MOVE   LINENBR,6
00410      GOTO   DELETE
00420 E2   MOVE   LINENBR,11
00430      GOTO   DELETE
00440 E3   MOVE   LINENBR,16
00450      GOTO   DELETE
00460 E4   MOVE   LINENBR,21
00470 DELETE ERASE  MODE=LINE,TYPE=DATA,LINE=LINENBR
00480      ADD     LINENBR,1
00490      ERASE  MODE=LINE,TYPE=DATA,LINE=LINENBR
00500      ADD     LINENBR,1
00510      ERASE  MODE=LINE,TYPE=DATA,LINE=LINENBR
00520      SUBTRACT LINENBR,2
00530      PRINTEXT LINE=LINENBR,SPACES=5
00540      TERMCTRL DISPLAY

```

Figure 14-69. \$FSEDIT (9)

The third and last scroll size option is MAX. With MAX, the scroll will be all the way to the top (PF2) or bottom (PF3) of the data set. After the MAX scroll operation, scroll size reverts to the normal scroll size for the mode in effect (normal scroll size for BROWSE mode is PAGE, and for EDIT mode is HALF).

While in BROWSE mode, the primary command LOCATE can be used to position the displayed data beginning at a specific statement number. In Figure 14-70, the primary command LOCATE 450 is entered into the command input area on the second line.

```

BROWSE - EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>> LOCATE 450          SCROLL ==>> PAGE
***** ***** TOP OF DATA *****
00010 XMLSTAT PROGRAM      START
00020 IOCB1      IOCB      NHIST=0
00030 IOCB2      IOCB      SCREEN=STATIC
00040          ATTNLIST    (END,OUT,SPF,STATIC)
00050 START      ENQT      IOCB1
00060          PRINTTEXT  'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00070          PRINTTEXT  'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00080          PRINTTEXT  'THE PROGRAM'
00090          PRINTTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00100          PRINTTEXT  'BRING UP THE ENTRY SCREEN'
00110          DEQT
00120 CHECK      WAIT      ATTNECB,RESET
00121          IF        (ATTNECB,EQ,1),GOTO,EMDIT
00140 ENTRY      ENQT      IOCB2
00150          ERASE     MODE=SCREEN,TYPE=ALL
00160          TERMCTRL  BLANK
00170          PRINTTEXT  'ENTER KEY = PAGE COMPLETE',LINE=1
00180          PRINTTEXT  '      PF1 = DELETE ENTRY 1'
00190          PRINTTEXT  '      PF2 = DELETE ENTRY 2'
00200          PRINTTEXT  '      PF3 = DELETE ENTRY 3      ',SKIP=1
00210          PRINTTEXT  '      PF4 = DELETE ENTRY 4'

```

Figure 14-70. \$FSEDIT (10)

When the enter key is pressed, the screen in Figure 14-71 will be displayed starting with statement 450.

```

BROWSE - EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>>
00450          GOTO      DELETE
00460 E4        MOVE     LINENBR,21
00470 DELETE    ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
00480          ADD      LINENBR,1
00490          ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
00500          ADD      LINENBR,1
00510          ERASE     MODE=LINE,TYPE=DATA,LINE=LINENBR
00520          SUBTRACT  LINENBR,2
00530          PRINTTEXT  LINE=LINENBR,SPACES=5
00540          TERMCTRL  DISPLAY
00550          GOTO      WAITONE
00560 READ      QUESTION  'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP
00570          ERASE     MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
00580          ERASE     MODE=SCREEN,LINE=6
00590          PRINTTEXT  LINE=6,SPACES=5
00600          TERMCTRL  DISPLAY
00610          GOTO      WAITONE
00620 CLEANUP   ERASE     MODE=SCREEN,TYPE=ALL
00630          DEQT
00640          GOTO      START
00650 ENDIT    PROGSTOP
00660          DATA     X'5050'

```

Figure 14-71. \$FSEDIT (11)

The "FIND" primary command performs the same type of positioning function using a text string instead of a statement number. In Figure 14-72 the command, FIND /ENDIT P/FIRST, is entered in the command input area.

The FIRST option means look for the text string beginning with the first statement in the data set. If FIRST is not specified, the search will begin with the first statement of the currently displayed screen. In this example, because the current screen is also the top of the data set, both options have the same effect.

```

BROWSE - EDITWORK, FDY002 75( 270)----- COLUMNS 001 072
COMMAND INPUT ==> [FIND/ENDIT P/ FIRST] SCROLL ==> PAGE
***** TOP OF DATA *****
00010 XEMPLSTAT PROGRAM START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
00040 ATTNLIST (END,OUT,$PF,STATIC)
00050 START ENQT IOCB1
00060 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00070 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00080 PRINTTEXT ' THE PROGRAM'
00090 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00100 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00110 DEQT
00120 CHECK WAIT ATTNECB,RESET
00121 IF (ATTNECB,EQ,1),GOTO,ENDIT
00140 ENTRY ENQT IOCB2
00150 ERASE MODE=SCREEN,TYPE=ALL
00160 TERMCTRL BLANK
00170 PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00180 PRINTTEXT ' PF1 = DELETE ENTRY 1'
00190 PRINTTEXT ' PF2 = DELETE ENTRY 2'
00200 PRINTTEXT 'PF3 = DELETE ENTRY 3 ',SKIP=1
00210 PRINTTEXT 'PF4 = DELETE ENTRY $'

```

Figure 14-72. \$FSEDIT (12)

When the ENTER key is pressed, the screen in Figure 14-73 will be displayed. The first statement is the statement containing the text string defined in the FIND command. The cursor will be positioned under the first character of the target string.

```

BROWSE - EDITWORK, EDX002      75( 270)----- CHARACTERS FOUND
COMMAND INPUT ==>                SCROLL ==> PAGE
00650 ENDIT      PROGSTOP
00660            DATA      X'5050'
00670 DASHES     DATA      80C'- '
00680 OUT        POST       ATTNECB,1
00690            ENDATTN
00700 STATIC     POST       ATTNECB,-1
00710            ENDATTN
00720 ATTNECB    ECB
00730 LINENBR    DATA      F'0'
00740            ENDPROG
00750            END
***** ***** BOTTOM OF DATA *****

```

Figure 14-73. \$FSEDIT (13)

If you want to find more than one occurrence of the same text string, the FIND command does not have to be reentered for each search. The first occurrence of the text string will be displayed as already illustrated. If PF4 is pressed, the search will continue. Each time the string is found, the statement containing the string will be displayed at the top of a new screen. Each time PF4 is pressed the search will continue, until the end of the data set is reached.

LOCATE, FIND, and MENU are the only primary commands recognized by BROWSE mode. MENU brings up the Primary Option Menu, shown in Figure 14-61.

### Option 7: MERGE

Option 7 allows you to combine (merge) two or more source data sets in the same edit work area. To demonstrate this option, a portion of the set of source statements created earlier (Figure 14-66) and stored in data set MRGDATA (Figure 14-65) will be merged with the current contents of the work area.

When option 7 is entered, you will be prompted on the lower half of the screen as shown in Figure 14-74. With the responses shown, statements 100 through 180 of data set MRGDATA will be merged into the present contents of the work data set following statement 30.

```

-----7--- $FSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

1 BROWSE - DISPLAY DATASET
2 EDIT   - CREATE OR CHANGE DATASET
3 READ   - READ DATASET FROM HOST/NATIVE
4 WRITE  - WRITE DATASET TO HOST/NATIVE
5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
6 LIST   - PRINT DATASET ON SYSTEM PRINTER
7 MERGE  - MERGE DATA FROM A SOURCE DATASET
8 END    - TERMINATE $FSEDIT
9 HELP   - DISPLAY TUTORIAL

-----
MERGE DATA FROM (NAME,VOLUME): MRGDATA,EDX002  LINES- 1ST LAST 100 180
ADD AFTER LINE #: 30

```

Figure 14-74. \$FSEDIT (14)

*Option 2: EDIT*

When option 2 is entered, the screen in Figure 14-75 is displayed. Notice that the merged statements have been inserted, and the entire data set renumbered.

```

EDIT --- EDITWORK,EDX002      84( 270)----- COLUMNS 001 072
COMMAND INPUT ==> [CHANGE /END,/QUIT,/FIRST]          SCROLL ==>HALF
***** TOP OF DATA *****
00010 XMPSTAT PROGRAM START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
00040 * * * * *
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130 ATTNLIST (END,OUT,$PF,STATIC)
00140 START ENQT IOCB1
00150 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170 PRINTTEXT ' THE PROGRAM'
00180 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00200 DEQT
00210 CHECK WAIT ATTNECB,RESET

```

Figure 14-75. \$FSEDIT (15)

In addition to LOCATE, FIND, and MENU, EDIT mode recognizes the CHANGE, RENUM, and RESET primary commands. In Figure 14-75, the primary command "CHANGE /END/QUIT/FIRST" is entered in the command input field. This command will look for the first occurrence of the text string END, starting with the first statement in the data set (FIRST). If NEXT is entered, the search would begin with the first statement on the current screen (the two statements have the same results in this example). When the text string END is found, it will be replaced with the text string QUIT. The first occurrence of END is in the ATTNLIST statement, at statement number 130 (Figure 14-75). In Figure 16-17, the ENTER key has been pressed, END has been changed to QUIT, and the first line displayed is the line the change occurred in. By pressing PF5, the CHANGE command can be repeated, with the search beginning with statement 130.

```

EDIT --- EDITWORK,EDX002      84( 270)----- TEXT CHANGED
COMMAND INPUT ==>                SCROLL ==HALF
00130      ATTNLIST (QUIT,OUT,$PF,STATIC)
00140 START  ENQT      IOCB1
00150      PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160      PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTTEXT ' THE PROGRAM'
00180      PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00200      DEQT
00210 CHECK  WAIT      ATTNECB,RESET
00220      IF          (ATTNECB,EQ,1),GOTO,ENDIT
00230 ENTRY  ENQT      IOCB2
00240      ERASE      MODE=SCREEN,TYPE=ALL
00250      TERMCTRL  BLANK
00260      PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00270      PRINTTEXT ' PF1 = DELETE ENTRY 1'
00280      PRINTTEXT ' PF2 = DELETE ENTRY 2'
00290      PRINTTEXT 'PF3 = DELETE ENTRY 3',SKIP=1
00300      PRINTTEXT 'PF4 = DELETE ENTRY 4'
00310      PRINTTEXT DASHES,PROTECT=YES,LINE=3
00320      PRINTTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
00330      PRINTTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00340 HDR    PRINTTEXT DASHES,PROTECT=YES,LINE=5

```

Figure 14-76. \$FSEDIT (16)

If you want to change every occurrence of a text string in the entire work area, ALL should be entered in place of FIRST or NEXT.

When in EDIT mode, changes to the displayed data may be entered directly onto the screen. In Figure 14-77, the QUIT in statement 130 has been changed back to END by overtyping.

```

EDIT --- EDITWORK, EDX002      84 ( 270)----- TEXT CHANGED
COMMAND INPUT ==>             SCROLL ==>
00130      ATTNLIST (END,OUT,$PF,STATIC)
00140 START  ENQT   IOCB1
00150      PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160      PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTTEXT ' THE PROGRAM'
00180      PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00200      DEQT
00210 CHECK  WAIT   ATTNECB,RESET
00220      IF      (ATTNECB,EQ,1),GOTO,ENDIT
00230 ENTRY  ENQT   IOCB2
00240      ERASE   MODE=SCREEN,TYPE=ALL
00250      TERMCTRL BLANK
00260      PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00270      PRINTTEXT ' PF1 = DELETE ENTRY 1'
00280      PRINTTEXT ' PF2 = DELETE ENTRY 2'
00290      PRINTTEXT 'PF3 = DELETE ENTRY 3',SKIP=1
00300      PRINTTEXT 'PF4 = DELETE ENTRY 4'
00310      PRINTTEXT DASHES,PROTECT=YES,LINE=3
00320      PRINTTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
00330      PRINTTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00340 HDR    PRINTTEXT DASHES,PROTECT=YES,LINE=5

```

Figure 14-77. \$FSEDIT (17)

The statements in the work data set may be renumbered using the RENUM primary command. In Figure 14-78, the RENUM command is used to renumber the data set in increments of 5, with the first statement assigned a statement number of 1.

```

EDIT --- EDITWORK, EDX002      84 ( 270)----- COLUMNS 001 07ALF
COMMAND INPUT ==>             SCROLL ==>
***** RENUM 1 5 *****
***** TOP OF DATA *****
00010 XPLSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040 *****
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 *****
00130      ATTNLIST (END,OUT,$PF,STATIC)
00140 START  ENQT   IOCB1
00150      PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=13,LINE=1
00160      PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTTEXT ' THE PROGRAM'
00180      PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00200      DEQT
00210 CHECK  WAIT   ATTNECB,RESET

```

Figure 14-78. \$FSEDIT (18)

Figure 14-79 is the resulting display, after the ENTER key has been pressed.

```

EDIT --- EDITWORK, EDX002      84( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00001 XMPLSTAT PROGRAM  START
00006 IOCB1   IOCB     NHIST=0
00011 IOCB2   IOCB     SCREEN=STATIC
00016 * * * * *
00021 *
00026 * MERGE DATA
00031 * MERGE DATA
00036 * MERGE DATA
00041 * MERGE DATA
00046 * MERGE DATA
00051 *
00056 * * * * *
00061          ATTNLIST (END,OUT,$PF,STATIC)
00066 START    ENQT     IOCB1
00071          PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00076          PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00081          PRINTEXT ' THE PROGRAM'
00086          PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00091          PRINTEXT ' BRING UP THE ENTRY SCREEN'
00096          DEQT
00101 CHECK    WAIT     ATTNECB,RESET

```

Figure 14-79. \$FSEDIT (19)

The RESET primary command is used in conjunction with the EDIT mode line commands, and will be illustrated later.

*Edit Mode Line Commands*

In addition to modification of text strings using the CHANGE primary command, and the modification of any displayed data on the screen by overtyping, EDIT mode also allows whole lines, or blocks of lines to be manipulated, using the EDIT mode line commands. For example, the INSERT (I) command allows a new line to be inserted between existing lines. In Figure 14-80, an "I" is entered to the left of statement 40, indicating that the operator wishes to insert between statement 40 and 50.

```

EDIT --- EDITWORK, EDX002      84( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040 * * * * *
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130          ATTHLIST (END,OUT,$PF,STATIC)
00140 START  ENQT      IOCB1
00150          PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160          PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170          PRINTEXT ' THE PROGRAM'
00180          PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190          PRINTEXT ' BRING UP THE ENTRY SCREEN'
00200          DEQT
00210 CHECK  WAIT      ATTHECB,RESET

```

Figure 14-80. \$FSEDIT (20)

When ENTER is pressed, the screen comes back as pictured in Figure 14-81, with the insert line displayed, and the cursor in the first character position, ready for entry.

```

EDIT --- EDITWORK, EDX002      84( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040 * * * * *
.....
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130          ATTHLIST (END,OUT,$PF,STATIC)
00140 START  ENQT      IOCB1
00150          PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160          PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170          PRINTEXT ' THE PROGRAM'
00180          PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190          PRINTEXT ' BRING UP THE ENTRY SCREEN'
00200          DEQT

```

Figure 14-81. \$FSEDIT (21)

When the insert line is complete, the operator presses the ENTER key, the new line is assigned a statement number, and another insert line is readied (Figure 14-82).

```

EDIT --- EDITWORK, EDX002      85( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPLSTAT PROGRAM  START
00020 IOCB1  IOCB  NHIST=0
00030 IOCB2  IOCB  SCREEN=STATIC
00040 * * * * *
00041 * INSERT SINGLE LINE
.....
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130      ATTNLIST  (END,OUT,$PF,STATIC)
00140 START      ENQT  IOCB1
00150      PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160      PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTEXT " THE PROGRAM"
00180      PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTEXT " BRING UP THE ENTRY SCREEN"

```

Figure 14-82. \$FSEDIT (22)

The operation terminates when ENTER is pressed with no characters entered on the insert line.

The INSERT BLOCK (II) command generates a block of 21 insert lines. In Figure 14-83 the "II" to the left of statement 50 indicates the operator wants to generate the insert block following statement 50.

```

EDIT --- EDITWORK, EEX002      85( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPLSTAT PROGRAM  START
00020 IOCB1  IOCB  NHIST=0
00030 IOCB2  IOCB  SCREEN=STATIC
00040 * * * * *
II00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130      ATTNLIST  (END,OUT,$PF,STATIC)
00140 START      ENQT  IOCB1
00150      PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160      PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTEXT " THE PROGRAM"
00180      PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTEXT " BRING UP THE ENTRY SCREEN"
00200      DEQT

```

Figure 14-83. \$FSEDIT (23)

When ENTER is pressed, the screen in Figure 14-84 is displayed.

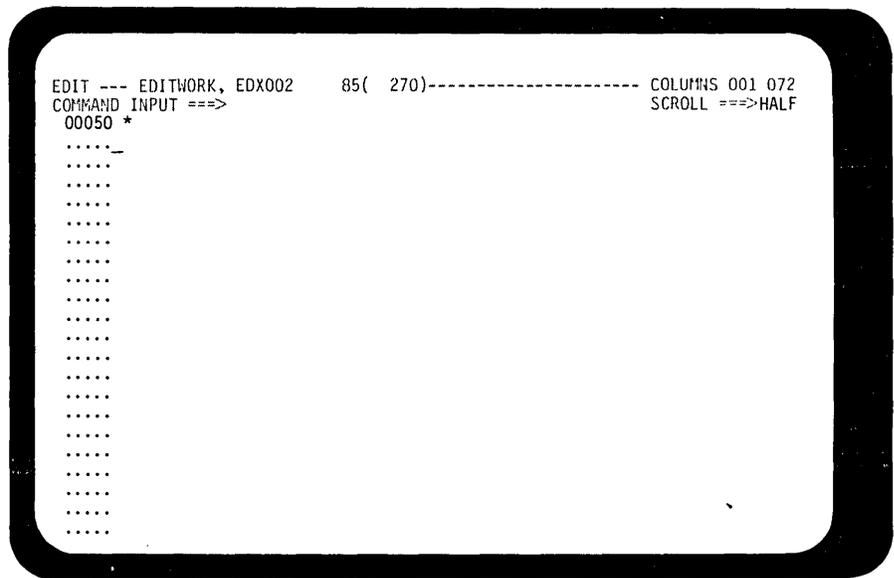


Figure 14-84. SFSEDIT (24)

The operator may now fill in the screen as required, without pressing ENTER for each line (Figure 14-85).

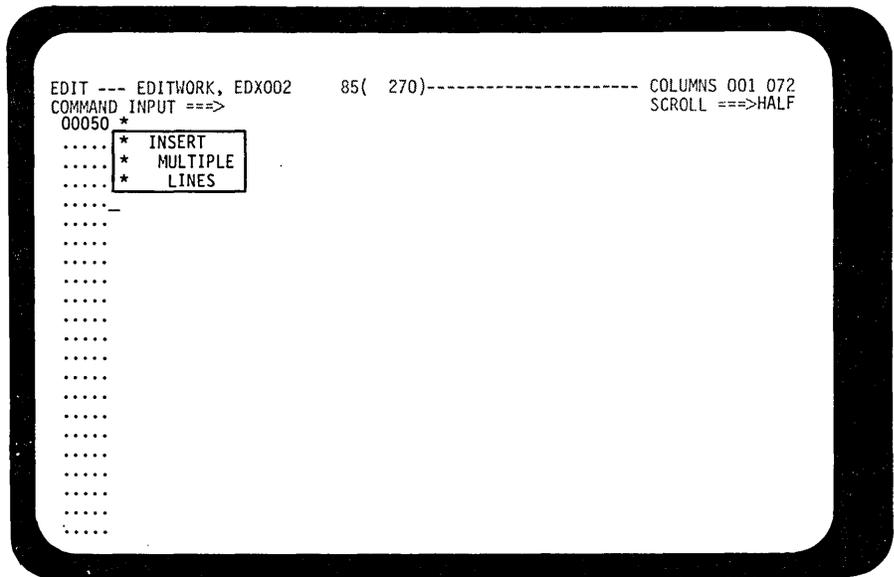


Figure 14-85. SFSEDIT (25)

When as much data as desired has been entered, the ENTER key is pressed.

Unused insert lines are removed, the insert lines used are assigned statement numbers, and the screen appears as shown in Figure 14-86.

```

EDIT --- EDITWORK, EDX002      88( 270)----- COLUMNS:001 072
COMMAND INPUT ==>                SCROLL ==>HALF
00050 *
00051 * INSERT
00052 * MULTIPLE
00053 * LINES
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 *
00120 * * * * *
00130      ATTNLIST (END,OUT,$PF,STATIC)
00140 START  ENQT   IOCB1
00150      PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00160      PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00170      PRINTTEXT ' THE PROGRAM'
00180      PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00190      PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00200      DEQT
00210 CHECK  WAIT   ATTNECB,RESET
00220      IF      (ATTNECB,EQ,1),GOTO,ENDIT
00230 ENTRY  ENQT   IOCB2

```

Figure 14-86. \$FSEDIT (26)

The MOVE (M) line command will move a line from one location in the work data set to another. In Figure 14-87, an "M" is entered to the left of the line to be moved, statement 50. The "A" at statement 140 specifies the destination of the MOVE as after line 140.

```

EDIT --- EDITWORK, EDX002      88( 270)----- DATA RENUMBERED
COMMAND INPUT ==>                SCROLL ==>HALF
***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  NHIST=0
00030 IOCB2  SCREEN=STATIC
00040 * * * * *
M00050 * INSERT SINGLE LINE
00060 *
00070 * INSERT
00080 * MULTIPLE
00090 * LINES
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
A00140 * MERGE DATA
00150 *
00160 * * * * *
00170      ATTNLIST (END,OUT,$PF,STATIC)
00180 START  ENQT   IOCB1
00190      PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00200      PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00210      PRINTTEXT ' THE PROGRAM'

```

Figure 14-87. \$FSEDIT (27)

Figure 14-88 is the screen displayed after ENTER is pressed. The line is moved, and the data set renumbered.

```

EDIT --- EDITWORK, EDX002      88( 270)----- DATA RENUMBERED
COMMAND INPUT ==>                SCROLL ==> HALF
00130 * MERGE DATA
00140 * INSERT SINGLE LINE
00150 *
00160 * * * * *
00170      ATTNLIST (END,OUT,$PF,STATIC)
00180 START  ENQT   IOCB1
00190      PRINTX  'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00200      PRINTX  'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00210      PRINTX  ' THE PROGRAM'
00220      PRINTX  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00230      PRINTX  ' BRING UP THE ENTRY SCREEN'
00240      DEQT
00250 CHECK  WAIT   ATTNECB,RESET
00260      IF      (ATTNECB,EQ,1),GOTO,ENDIT
00270 ENTRY  ENQT   IOCB2
00280      ERASE  MODE=SCREEN,TYPE=ALL
00290      TERMCTRL BLANK
00300      PRINTX  'ENTER KEY = PAGE COMPLETE',LINE=1
00310      PRINTX  ' PF1 = DELETE ENTRY 1'
00320      PRINTX  ' PF2 = DELETE ENTRY 2'
00330      PRINTX  'PF3 = DELETE ENTRY 3'          ',SKIP=1
00340      PRINTX  'PF4 = DELETE ENTRY 4'

```

Figure 14-88. \$FSEDIT (28)

The MOVE BLOCK line command (MM) is illustrated in Figure 14-89. The MM to the left of statements 60 and 80 define the inclusive start and end points of a block of statements to be moved. The B defines the destination of the block as before statement 150. (Either A or B can be used with M and MM.)

```

EDIT --- EDITWORK, EDX002      88( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
00020 IOCB1  IOCB   NHIST=0
00030 IOCB2  IOCB   SCREEN=STATIC
00040 * * * * *
00050 *
MM 00060 * INSERT
MM 00070 * MULTIPLE
MM 00080 * LINES
MM 00090 * MERGE DATA
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * INSERT SINGLE LINE
B 00150 *
00160 * * * * *
00170      ATTNLIST (END,OUT,$PF,STATIC)
00180 START  ENQT   IOCB1
00190      PRINTX  'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00200      PRINTX  'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00210      PRINTX  ' THE PROGRAM'
00220      PRINTX  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00230      PRINTX  ' BRING UP THE ENTRY SCREEN'

```

Figure 14-89. \$FSEDIT (29)

After ENTER is pressed, the screen in Figure 14-90 is displayed.

```

EDIT --- EDITWORK, EDX002      88( 270)----- BLOCK -- DATA RENUMBERED
COMMAND INPUT ==>                                SCROLL ==>                                HALF

00110 * INSERT SINGLE LINE
00120 * INSERT
00130 * MULTIPLE
00140 * LINES
00150 *
00160 * * * * *
00170 * * * * *
00180 * * * * *
00180 START ENQT IOCB1
00190 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00200 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00210 PRINTTEXT ' THE PROGRAM'
00220 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00230 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00240 DEQT
00250 CHECK WAIT ATTNECB,RESET
00260 IF (ATTNECB,EQ,1),GOTO,ENDIT
00270 ENTRY ENQT IOBC2
00280 ERASE MODE=SCREEN,TYPE=ALL
00290 TERMCTRL BLANK
00300 PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00310 PRINTTEXT ' PF1 = DELETE ENTRY 1'
00320 PRINTTEXT ' PF2 = DELETE ENTRY 2'

```

Figure 14-90. \$FSEDIT (30)

The MOVE and MOVE BLOCK commands removed statements from one part of the work data set and placed them in another. The COPY (C) and COPY BLOCK (CC) line commands reproduce an exact copy of the designated statement(s) at another location in the data set without disturbing the original. In Figure 14-91, statement number 110 is to be copied after statement 40.

```

EDIT --- EDITWORK, EDX002      88( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                                SCROLL ==>                                HALF

***** ***** TOP OF DATA *****
00010 XPLSTAT PROGRAM START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
00040 * * * * *
00050 *
00060 * MERGE DATA
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 * INSERT SINGLE LINE
00120 * INSERT
00130 * MULTIPLE
00140 * LINES
00150 *
00160 * * * * *
00170 * * * * *
00180 * * * * *
00180 START ENQT IOCB1
00190 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00200 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00210 PRINTTEXT ' THE PROGRAM'

```

Figure 14-91. \$FSEDIT (31)

In Figure 14-92, the operation is complete (ENTER key has been pressed).

```

EDIT --- EDITWORK, EDX002      89( 270)----- DATA RENUMBERED
COMMAND INPUT ==>                SCROLL ==> HALF
00040 * * * * *
00050 * INSERT SINGLE LINE
00060 *
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 * MERGE DATA
00120 * INSERT SINGLE LINE
00130 * INSERT
00140 * MULTIPLE
00150 * LINES
00160 *
00170 * * * * *
00180          ATTNLIST (END,OUT,SPF,STATIC)
00190 START   ENQT     IOCB1
00200          PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00210          PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00220          PRINTTEXT ' THE PROGRAM'
00230          PEINTEXT  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00240          PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00250          DEQT

```

Figure 14-92. \$FSEDIT (32)

In Figures 14-93 and 14-94, the same operation is performed with the COPY BLOCK (CC) line command, copying statements 130 through 150.

```

EDIT --- EDITWORK, EDX002      89( 270)----- DATA RENUMBERED
COMMAND INPUT ==>                SCROLL ==> HALF
00040 * * * * *
A 00050 * INSERT SINGLE LINE
00060 *
00070 * MERGE DATA
00080 * MERGE DATA
00090 * MERGE DATA
00100 * MERGE DATA
00110 * MERGE DATA
00120 * INSERT SINGLE LINE
CC 00130 * INSERT
00140 * MULTIPLE
CC 00150 * LINES
00160 *
00170 * * * * *
00180          ATTNLIST (END,OUT,SPF,STATIC)
00190 START   ENQT     IOCB1

```

Figure 14-93. \$FSEDIT (33)

```

EDIT --- EDITWORK, EDX002      92( 270)----- BLOCK -- DATA RENUMBERED
COMMAND INPUT ==>                SCROLL ==>HALF

00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *
00200 * * * * *
00210 * * * * * ATTNLIST (END,OUT,$PF,STATIC)
00220 START ENQT IOCB1
00230 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00240 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00250 PRINTTEXT ' THE PROGRAM'
00260 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2

```

Figure 14-94. \$FSEDIT (34)

When the INSERT LINE (I) and INSERT BLOCK (II) line commands were discussed (Figures 14-80 through 14-85), the I command resulted in the display of a blank insert line. This insert line is actually an insert mask, initialized to blanks. The insert mask may be displayed using the MASK line command. In Figure 14-95, the MASK command is typed in over the first four digits of the sequence number of statement 40. It does not matter what statement's sequence number is overtyped; the data on that line is not destroyed.

```

EDIT --- EDITWORK, EDX002      92( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
***** TOP OF DATA *****
00010 XMPSTAT PROGRAM START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
MASK *****
00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *
00200 * * * * *
00210 * * * * * ATTNLIST (END,OUT,$PF,STATIS)

```

Figure 14-95. \$FSEDIT (35)

When the ENTER key is pressed, the insert mask is displayed. As you can see in Figure 14-96, the insert mask is the line of blanks that was inserted every time you entered the I command.

```

EDIT --- EDITWORK, EDX002      92( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  IOCB     NHIST=0
00030 IOCB2  IOCB     SCREEN=STATIC
00040 * * * * *
MASK
00050 *  INSERT SINGLE LINE
00060 *  INSERT
00070 *  MULTIPLE
00080 *  LINES
00090 *
00100 *  MERGE DATA
00110 *  MERGE DATA
00120 *  MERGE DATA
00130 *  MERGE DATA
00140 *  MERGE DATA
00150 *  INSERT SINGLE LINE
00160 *  INSERT
00170 *  MULTIPLE
00180 *  LINES
00190 *
00200 * * * * *

```

Figure 14-96. \$FSEDIT (36)

(Notice that statement 40, whose sequence number was used for the MASK command input field, is intact.)

You can redefine the insert mask to be any character string you wish. In Figure 14-97, the mask has asterisks entered in the leading and ending character positions.

```

EDIT --- EDITWORK, EDX002      92( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  IOCB     NHIST=0
00030 IOCB2  IOCB     SCREEN=STATIC
00040 * * * * *
MASK *****
00050 *  INSERT SINGLE LINE
00060 *  INSERT
00070 *  MULTIPLE
00080 *  LINES
00090 *
00100 *  MERGE DATA
00110 *  MERGE DATA
00120 *  MERGE DATA
00130 *  MERGE DATA
00140 *  MERGE DATA
00150 *  INSERT SINGLE LINE
00160 *  INSERT
00170 *  MULTIPLE
00180 *  LINES
00190 *
00200 * * * * *

```

Figure 14-97. \$FSEDIT (37)

To get out of this insert mask display/definition mode, move the cursor to the primary command input area on the second line of the screen, type in the primary command RESET, and press ENTER.

The RESET primary command is also used to reset undesired but already entered line commands, and to reset error conditions resulting from improper use of line commands.

Now that the insert mask display has been RESET, a Line Insert command is entered (Figure 14-98).

```
EDIT --- EDITWORK, EDX002      92( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1  IOCB  NHIST=0
00030 IOCB2  IOCB  SCREEN=STATIC
00040 * * * * *
00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *
00200 * * * * *
00210          ATTNLIST (END,OUT,$PF,STATIC)
```

Figure 14-98. \$FSEDIT (38)

When the insert line appears, the line contains the redefined mask characters (Figure 14-99).

```

EDIT --- EDITWORK, EDX002      92( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPLSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040 * * * * *
00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
*****
.....
00100 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *
00200 * * * * *

```

Figure 14-99. \$FSEDIT (39)

Each time another insert line appears, the mask characters are displayed. You can enter data on top of them if desired, or in the blank areas between them, as in Figure 14-100.

```

EDIT --- EDITWORK, EDX002      95( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPLSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040 * * * * *
00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00091 ***** WITH THE INSERT MASK DEFINED, EACH TIME AN *****
00092 ***** INSERT LINE IS DISPLAYED, THE MASK CHARACTERS *****
00093 ***** ARE DISPLAYED ON THE SAME LINE. *****
.....
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES

```

Figure 14-100. \$FSEDIT (40)

The DELETE Line (D) and DELETE Block (DD) line commands remove statement(s) from the work data set. In Figure 14-101, the D command is entered to the left of line 50.

```

EDIT --- EDITWORK, EDX002      95( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
**** ***** TOP OF DATA *****
00010 XPLSTAT PROGRAM  START
00020 IOCB1   IOCB   NHIST=0
00030 IOCB2   IOCB   SCREEN=STATIC
00040 * * * * *
00050 * INSERT SINGLE LINE
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00091 ***** WITH THE INSERT MASK DEFINED, EACH TIME AN *****
00092 ***** INSERT LINE IS DISPLAYED, THE MASK CHARACTERS *****
00093 ***** ARE DISPLAYED ON THE SAME LINE. *****
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES

```

Figure 14-101. \$FSEDIT (41)

After the ENTER key is pressed, the screen in Figure 14-102 appears with line 50 deleted.

```

EDIT --- EDITWORK, EDX002      94( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==>HALF
**** ***** TOP OF DATA *****
00010 XPLSTAT PROGRAM  START
00020 IOCB1   IOCB   NHIST=0
00030 IOCB2   IOCB   SCREEN=STATIC
00040 * * * * *
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00091 ***** WITH THE INSERT MASK DEFINED, EACH TIME AN *****
00092 ***** INSERT LINE IS DISPLAYED, THE MASK CHARACTERS *****
00093 ***** ARE DISPLAYED ON THE SAME LINE. *****
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *

```

Figure 14-102. \$FSEDIT (42)

In Figure 14-103, the first statement of a block delete is defined with the DD command.

```

EDIT --- EDITWORK, EDX002      94( 270)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPSTAT PROGRAM  START
00020 IOCB1 IOCB      NHIST=0
00030 IOCB2 IOCB      SCREEN=STATIC
DD00040 * * * * *
00060 * INSERT
00070 * MULTIPLE
00080 * LINES
00090 *
00091 ***** WITH THE INSERT MASK DEFINED, EACH TIME AN *****
00092 ***** INSERT LINE IS DISPLAYED, THE MASK CHARACTERS *****
00093 ***** ARE DISPLAYED ON THE SAME LINE. *****
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *

```

Figure 14-103. \$FSEDIT (43)

The ending statement to be deleted is not displayed on this screen, so PF3 is pressed, scrolling down a half-page, to the screen displayed in Figure 14-104.

```

EDIT --- EDITWORK, EDX002      94( 270)----- BLOCK COMMAND INCOMPLETE
COMMAND INPUT ==>                SCROLL ==> HALF
00093 ***** ARE DISPLAYED ON THE SAME LINE. *****
00100 * MERGE DATA
00110 * MERGE DATA
00120 * MERGE DATA
00130 * MERGE DATA
00140 * MERGE DATA
00150 * INSERT SINGLE LINE
00160 * INSERT
00170 * MULTIPLE
00180 * LINES
00190 *
DD00200 * * * * *
00210 ***** ATTNLIST (END,OUT,$PF,STATIC) *****
00220 START ENQT IOCB1
00230 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00240 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00250 PRINTTEXT ' THE PROGRAM'
00260 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00270 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00280 DEQT
00290 CHECK WAIT ATTNECB,RESET
00300 IF (ATTNECB,EQ,1),GOTO,ENDIT

```

Figure 14-104. \$FSEDIT (44)

(The scope of the C, CC, M, MM, D, and DD line commands extends from the beginning to the end of the data in the work area, not just the data on the current screen.)

The end of the Delete Block is entered at statement 200 (Figure 14-104).

After the command is entered, the screen in Figure 14-105 is displayed. All statements merged, inserted, copied or moved during the course of this exercise have been deleted, and the data set is in the same state it was in when it was first read from SOURCE.

```

EDIT --- EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==>
00210      ATTNLIST (END,OUT,$PF,STATIC)
00220 START  ENQT      IOCB1
00230      PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00240      PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00250      PRINTEXT ' THE PROGRAM'
00260      PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00270      PRINTEXT ' BRING UP THE ENTRY SCREEN'
00280      DEQT
00290 CHECK  WAIT      ATTNECB,RESET
00300      IF          (ATTNECB,EQ,1),GOTO,ENDIT
00310 ENTRY  ENQT      IOCB2
00320      ERASE      MODE=SCREEN,TYPE=ALL
00330      TERMCTRL  BLANK
00340      PRINTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00350      PRINTEXT ' PF1 = DELETE ENTRY 1'
00360      PRINTEXT ' PF2 = DELETE ENTRY 2'
00370      PRINTEXT 'PF3 = DELETE ENTRY 3           ',SKIP=1
00380      PRINTEXT 'PF4 = DELETE ENTRY 4'
00390      PRINTEXT DASHES,PROTECT=YES,LINE=3
00400      PRINTEXT 'CLASS NAME:',LINE=4,PROTECT=YES,SPACES=32
00410      PRINTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00420 HDR    PRINTEXT DASHES,PROTECT=YES,LINE=5

```

Figure 14-105. \$FSEDIT (45)

The MENU primary command is entered in the command input field, and ENTER pressed.

```

EDIT --- EDITWORK, EDX002      75( 270)----- COLUMNS 001 072
COMMAND INPUT ==> MENU
00210      ATTNLIST (END,OUT,$PF,STATIC)
00220 START  ENQT      IOCB1
00230      PRINTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00240      PRINTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00250      PRINTEXT ' THE PROGRAM'
00260      PRINTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00270      PRINTEXT ' BRING UP THE ENTRY SCREEN'
00280      DEQT
00290 CHECK  WAIT      ATTNECB,RESET
00300      IF          (ATTNECB,EQ,1),GOTO,ENDIT
00310 ENTRY  ENQT      IOCB2
00320      ERASE      MODE=SCREEN,TYPE=ALL
00330      TERMCTRL  BLANK
00340      PRINTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00350      PRINTEXT ' PF1 = DELETE ENTRY 1'
00360      PRINTEXT ' PF2 = DELETE ENTRY 2'
00370      PRINTEXT 'PF3 = DELETE ENTRY 3           ',SKIP=1
00380      PRINTEXT 'PF4 = DELETE ENTRY 4'
00390      PRINTEXT DASHES,PROTECT=YES,LINE=3
00400      PRINTEXT 'CLASS NAME:',LINE=4,PROTECT=YES
00410      PRINTEXT 'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00420 HDR    PRINTEXT DASHES,PROTECT=YES,LINE=5

```

Figure 14-106. \$FSEDIT (46)

Option 8

The only Primary Option remaining to be discussed is option 8.



## OTHER UTILITY PROGRAMS

The following utility programs are used with system facilities not addressed as topics in this study guide.

### BSC Utilities

READING REFERENCE: IBM Series/1 Event Driven Executive Communications and Terminal Applications Guide (SC34-1705), "\$BSTRCE Utility Program," "\$BSCUT1 Utility Program," and "\$BSCUT2 Utility Program."

#### \$BSTRCE

This utility traces I/O on a specified BSC line, and stores the trace data in a data set on disk or diskette. The data set must be preallocated by the user, and the name supplied to the \$BSTRCE utility at the time the utility is loaded. Trace information includes condition codes, status words, data transferred, and other indicators/information associated with BSC I/O operation.

#### \$BSCUT1

Trace information written by \$BSTRCE is retrieved and formatted into an easily understood report by \$BSCUT1, and then directed to a specified terminal or print device.

#### \$BSCUT2

This utility is a BSC exerciser, used to test the BSC hardware adapter, and the match between the actual hardware configuration and what has been specified in the BSCLINE system configuration statement. Several BSC access method commands may be invoked to exercise various hardware/system software combinations.

## DISPLAY PROCESSOR (GRAPHICS) UTILITIES

READING REFERENCE: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "Graphics Utilities."

The Display Processor facility allows the user to generate, store, and display information in graphic or report format. The information is contained in a data base created expressly for, and utilizing, data organization and data formatting conventions unique to the Display Processor. Display Processor support consists of three utility programs, which are used to create/maintain the data base, create or alter data members, or display a graphic or report data member.

### \$DIUTIL

This utility provides all data base maintenance functions for the Display Processor data base, including initialization, member deletion/allocation, data base compression, and member/data base copy.

### \$DICOMP

A member within the Display Processor data base is called a display profile. This utility allows the operator to compose a display profile, or to alter (maintain) existing display profiles.

### \$DIINTR

A completed display profile (data base member) is made up of coded information representing an image or report. The \$DIINTR utility retrieves a specified display profile, interprets the coded commands/data it contains, and displays the resulting image.

*Note:* Terminals used as graphics devices must have ASCII point-to-point vector graphics capability.

## HOST PROGRAM PREPARATION UTILITIES

READING REFERENCE: IBM Series/1 Event Driven Executive Communications and Terminal Applications Guide (SC34-1705), "\$HCFUT1 Utility Program."

## **\$HCFUT1**

When program preparation is performed on a host System/370, the Host Communications Facility IUP (5796-PGH) must be installed on the host system. On the Series/1 side the \$HCFUT1 utility program is used.

\$HCFUT1 is the basic Event Driven Executive utility program used to transfer data sets associated with program preparation between the Series/1 and a host system. The four functions available are;

1. READ a source/object data set from a host into a Series/1 data set
2. WRITE a Series/1 source/object data set to a host data set
3. SUBMIT a program preparation job to the host job stream
4. SET/FETCH/RELEASE a record in the host System Status data set

## **\$EDIT1/\$UPDATEH**

These are the host preparation equivalents of the native preparation text editing and object module formatting utilities \$EDIT1N and \$UPDATE. They differ from the native versions only in the commands used to store and retrieve source and object module data sets. For the native versions, any operation involving a data set transfer (READ/SAVE/RP) requires that both the from and to data sets be resident on the Series/1. With the "host prep" versions, both will be resident on the host.

\$EDIT1 and \$UPDATEH invoke the READ and WRITE (also SUBMIT for \$EDIT1) functions of \$HCFUT1 without the operator's having to load \$HCFUT1 explicitly. If the operator does load \$HCFUT1 and uses it for the necessary data set transfers, then the editing/formatting operations would be done with \$EDIT1N and \$UPDATE.

*Note:* \$FSEDIT, the full screen text edit utility, includes host prep data set transfer functions in its normal command menu; no separate version for host program preparation is required.

## **\$RJE2780/\$RJE3780**

READING REFERENCE: IBM Series/1 Event Driven Executive Communications and Terminal Application Guide (SC34-1705), "\$RJE2780 and \$RJE3780 Utility Programs" and "\$PRT2780 and \$PRT3780 Utility Programs."

These utilities provide an alternative method of transferring data sets between a Series/1 and a host program preparation system. The \$RJE2780 and \$RJE3780 simulate the IBM 2780 and IBM 3780 Remote Job Entry stations. Using the Series/1 BSC capability, \$RJE2780 and \$RJE3780 interface to System/360 or System/370 systems with the Remote Job Entry facility installed (5796-PGH not required).

## **\$PRT2780/\$PRT3780**

These utilities print the RJE printer output spool files created when \$RJE2780/\$RJE3780 is used with the spooling option invoked.

## **\$DEBUG**

**READING REFERENCE:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$DEBUG Utility Program."

**\$DEBUG** is the Event Driven Executive online debugging utility. **\$DEBUG** may be used to debug any program instructions that execute as a task, including instructions written in Series/1 assembler language. **\$DEBUG** capabilities include setting/resetting of breakpoints and trace ranges; display and modification of storage locations; Series/1 hardware registers, and task software registers; and alteration of task execution sequence.



This page intentionally left blank.



## Section 15. System Installation

### OBJECTIVES:

After completing this section, the student should be able to generate a tailored supervisor for a given sample configuration, using the programs/utilities provided in the Event Driven Executive Program Preparation Facility.

### MACHINE READABLE MATERIAL

The Event Driven Executive software offering for the Series/1 is comprised of five separately orderable programs:

1. 5719-XS3 Event Driven Executive Basic Supervisor and Emulator
2. 5719-UT5 Event Driven Executive Utility Programs
3. 5719-XX4 Event Driven Executive Program Preparation Facility
4. 5719-LM7 Event Driven Executive Macro Library
5. 5740-LM4 Event Driven Executive Macro Library/Host

#### 5719-XS3 Basic Supervisor and Emulator

Diskette XS3001 contains the supplied starter supervisor and the necessary utilities to install the product.

Diskette XS3002 contains supervisor object modules used during the system generation process.

Diskette XS3003 contains object modules that support various system functions.

#### 5719-UT5 System Utility Programs

Diskettes UT5001-2 contain the link editor, the utility programs and session manager programs.

#### 5719-XX4 Program Preparation Facility

Diskette XX4001 contains the EDX program preparation modules.

Diskette XX4002 contains copy modules (DSOPEN, SETEOD, etc.) for inclusion in user application programs.

## 5719-LM7 Macro Library

5719-LM7 is a library containing source macro definitions for the Event Driven Executive instruction set and system configuration statements. This macro library is used when assembling Event Driven Executive programs using the Series/1 Macro Assembler, 5719-ASA. It is not required, and *cannot be used* by the Event Driven language assembler \$EDXASM, as \$EDXASM is not a macro assembler.

## 5740-LM4 Macro Library/Host

This library is distributed on tape for installation on host S/370s, which will be used to assemble Event Driven Executive programs. Included are source macro definitions for the Event Driven Executive instruction set, as well as procedures/JCL prototypes to aid in host installation.

## STARTER SYSTEM INSTALLATION OVERVIEW

*Note:* This discussion, and the "USER SYSTEM GENERATION" topic which follows, will be limited to the Basic Supervisor and Emulator, the Utility Programs, and the Program Preparation Facility. The Macro Library (5719-LM7) and Macro Library/Host (5740-LM4) licensed programs will not be addressed.

Installation is supported for:

- 4962 Model 1 or 2 (9.3MB)
- 4962 Model 1F or 2F (9.3MB with fixed heads)
- 4962 Model 3 or 4 (13.9MB)
- 4963 Model 1 (29MB)
- 4963 Model 1F (23MB with fixed heads)
- 4963 Model 2 (64MB)
- 4963 Model 2F (58MB with fixed heads)

An Event Driven Executive supervisor, to IPL, must reside in a data set named \$EDXNUC. As shipped from PID, diskette XS3001 contains the starter supervisor in a data set named \$EDXNUC. The first step in starter system installation requires that an IPL of the starter supervisor from diskette XS3001 be performed. Therefore, the Series/1 on which the starter system is being installed

MUST HAVE EITHER A 4964 Diskette Unit at hardware address X'02'

OR A 4966 Diskette Magazine Unit at hardware address X'22'

wired as an IPL device (PRIMARY or ALTERNATE).

*Note:* If a 4966 is the IPL device, an IPL can be performed from diskette slot 1 only.

The object of starter system installation is to transfer the programs and utilities supplied on the PID diskettes to a disk device. Among the programs transferred is \$EDXNUC, the starter supervisor itself, so that an IPL can be performed from disk rather than diskette. Therefore, the Series/1 on which the starter system is being installed

MUST HAVE EITHER A 4962 (any model) installed at hardware address X'03'

OR A 4963 (any model) installed at hardware address X'48'

wired as an IPL device (PRIMARY or ALTERNATE). In addition, the supplied supervisor assumes certain terminal device availability and hardware address assignments. The Series/1 on which the starter system is being installed

MUST HAVE EITHER A TTY device at hardware address '00'

OR A 4978 or 4979 Display at hardware address '04'

AND MAY HAVE A 4974 Matrix Printer at hardware address X'01'

## INSTALLING THE STARTER SYSTEM

The procedures and instructions for installing the starter system received from PID are contained in the Program Directory which is shipped with the licensed program diskettes. As new versions or modification levels of the system are released, details of the installation process may change. This discussion is therefore limited to the major steps involved.

*Step 1: IPL the Starter Supervisor from XS3001.* When the starter supervisor is first loaded (IPL) and goes into execution, it searches for a 4963 Disk at hardware address x'48'. If there is no 4963 Disk the supervisor searches for a 4962 Disk at address x'03'. If found, it reads the device ID (which contains information about the device) and alters the device data block (DDB) for the associated disk.

*Step 2: Initialize Logical Volumes.* Before copying any data sets, a volume directory must be written on disk, volumes allocated and directories created. Review the example in the \$INITDSK portion of the Utilities Section of this document. See the Program Directory for recommended volume and directory sizes.

*Step 3: Copy Starter Supervisor, Utility Programs and System Support Modules.* The utility program \$COPYUT1 is now used to copy the Starter Supervisor \$EDXNUC on XS3001 to \$EDXNUC on EDX002. Some of the system utility programs are also copied to EDX002. The system support modules are copied from diskette XS3003 to volume ASMLIB on disk.

*Step 4: IPL Starter System from Disk and Complete Installation.* The IPL SOURCE switch can now be set to IPL from disk, and the starter system again loaded, this time from the disk IPL volume.

EDX002. \$COPYUT1 is again loaded, this time from EDX002 to which it was copied in the previous step. \$COPYUT1 is used to copy the remaining system programs on the PID diskettes to the various libraries on disk.

Source Diskette	Target Volume	Description
UT5001-2	EDX002	Remaining utilities and Session Manager modules
XX4001	ASMLIB	Program Preparation modules
XX4002	ASMLIB	Copy Code Modules

The Starter Supervisor as supplied by IBM supports the following:

64 KB Storage

4962 Disk at address 03

or

4963 Disk at address 48

4964 Diskette Unit at address 02

4962 Diskette Magazine at address 22

4978 or 4979 Display at address 04

TTY Device at address 0

3101 Model 2X in block mode

at address 08 via asynchronous communications single line adapter

at address 60 via asynchronous communications multiline adapter

at address 68 via programmable communication adapter

The Starter Supervisor does *not* support:

Timers

Sensor I/O

Communications

Interactive Debug

Magnetic Tape

Series/1 to Series/1

General Purpose Interface Bus (GPIB)

Spooling

Floating Point Arithmetic

Users who have different requirements from those provided by the Starter Supervisor must generate a tailored system that will satisfy their needs.

## USER SYSTEM GENERATION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive System Guide (SC34-1702) "System Generation." (ALL)

### SYSGEN OVERVIEW

Creating a supervisor tailored to a specific user configuration consists of the following tasks:

1. Creating a set of system configuration statements reflecting the configuration of the system that the supervisor being generated is to run on.
2. Selecting the supervisor object modules that are required to support the desired I/O devices and system features.
3. Assembling the system configuration statements created in Step 1, above.
4. Link editing the object module produced by the assembly in Step 3 with the supervisor object modules selected in Step 2 to produce a tailored supervisor.

In order to demonstrate how these tasks may be accomplished, the remainder of this section will go through each step of an actual system generation.

### Allocate Required Data Sets

After completion of starter system installation, the system programs are installed, but no user-allocated data sets have yet been defined. The system generation process requires the use of several system utility/program preparation programs that require data sets for use as work areas or input/output files. These data sets must be allocated by the user before SYSGEN can proceed. Data set allocation is done with the \$DISKUT1 utility program.

> \$L \$DISKUT1  
\$DISKUT1 37P, LP= 5700

USING VOLUME EDX002

COMMAND (?): AL EDITWORK 200 D ← 1  
EDITWORK CREATED

COMMAND (?): AL ASMOBJ 250 D ← 2  
ASMOBJ CREATED

COMMAND (?): AL ASMWORK 250 ← 2  
ASMWORK CREATED

COMMAND (?): AL SUPVLINK 600 D ← 3  
SUPVLINK CREATED

COMMAND (?): AL LEWORK1 400 D ← 4  
LEWORK1 CREATED

COMMAND (?): AL LEWORK2 150 D ← 4  
LEWORK2 CREATED

COMMAND (?): END

\$DISKUT1 ENDED

Figure 15-1. Allocate data sets

- 1** EDITWORK is the name of a work file that will be required by \$EDIT1N or \$FSEDIT text editing utilities.
- 2** These data sets are used by the assembler program \$EDXASM. ASMOBJ is the data set in which the object module output of the assembler will be stored, and ASMWORK is an assembler work file. *Note:* In the Program Directory, it is suggested that you assemble the sample program CALCSRC to verify starter system installation. If you performed that step, ASMWORK and ASMOBJ have already been allocated, and need not be allocated here.
- 3** SUPVLINK is the data set where the link editor, \$LINK, will store the linked object module output from the supervisor link edit.
- 4** LEWORK1 and LEWORK2 are \$LINK work data sets.

## EDIT SYSTEM CONFIGURATION STATEMENTS

Before proceeding, you must know the configuration of the system you intend to run the supervisor on, and what features you want to support. You can generate a supervisor for a system other than the one used for SYSGEN, but for this discussion, assume the tailored supervisor being built is for the system you are now running on.

## \$IOTEST

One of the operands you must specify in all of the system configuration statements defining I/O devices is the device hardware address. The system utility program \$IOTEST can be used to find out which devices are installed on your system and what their addresses are (Figure 15-2).

```
> $L $IOTEST
$IOTEST      32P, LP= 8F00

COMMAND (?): LD

ACTUAL SERIES/1 HARDWARE CONFIGURATION

ADDRESS      DEVICE TYPE

00 = TELETYPEWRITER ADAPTER
01 = 4974 PRINTER
02 = 4964 DISKETTE UNIT
03 = 4962 DISK MDL3
04 = 4979 DISPLAY STATION
06 = 4978 DISPLAY STATION
08 = SINGLE LINE ACCA
21 = 4973 PRINTER
40 = TIMER FEATURE
41 = TIMER FEATURE
```

Figure 15-2. \$IOTEST LD

In Figure 15-3 below, the LS command is used to list the hardware devices supported by the starter supervisor under which \$IOTEST is running.

```
COMMAND (?): LS

HARDWARE DEVICES SUPPORTED BY THIS SUPERVISOR

ADDRESS      DEVICE TYPE

00 = TELETYPEWRITER ADAPTER
01 = 4973 PRINTER
02 = 4964 DISKETTE UNIT
03 = 4962 DISK MDL3
04 = 4978 DISPLAY STATION
08 = SINGLE LINE ACCA MODE 3101B
22 = 4966 DISKETTE MAGAZINE UNIT
60 = FOUR LINE ACCA MODE 3101B
68 = FOUR LINE ACCA MODE 3101B

COMMAND (?): END
```

Figure 15-3. \$IOTEST LS

By comparing Figures 15-2 and 15-3, you can see that the starter supervisor does not support the 4978 Display at address 06, the 4973 printer at address 21 or the timers at address 40 and 41.

After the tailored system generation is complete and the new supervisor is loaded, the LS command of \$IOTEST should result in a printout of supervisor-supported devices and address assignments, which matches the LD command output shown in Figure 15-2.

Now you are ready to build a system configuration statement source file that reflects the I/O and system features you wish to support. This file can be created using either \$EDIT1N or \$FSEDIT.

During the installation procedure, a data set reflecting the configuration statements used in generation the starter supervisor was copied to disk. The data set is \$EDXDEF on volume ASMLIB. If you load the Text Editor, read the data set and list it, the contents would be as shown in Figure 15-4.

```

00010 $EDXDEF  CSECT
00020          DATA  F'0'
00030 *
00040 * EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
00050 *
00060 * THE FOLLOWING DEFINES THE STARTER SUPERVISOR AS SHIPPED ON THE
00070 * DISKETTE LABELED XS3001. FOR COMPLETE DESCRIPTIONS OF THESE
00080 * STATEMENTS OR ANY OTHER SYSTEM DEFINITION STATEMENTS, REFER TO
00090 * THE EDX VERSION 3 SYSTEM GUIDE: SC34-1702
00100 *
00110          SYSTEM STORAGE=64,MAXPRG=10,PARTS=32
00120          DISK   DEVICE=4963-23,ADDRESS=48
00130          DISK   DEVICE=4964,ADDRESS=02
00140          DISK   DEVICE=4966,ADDRESS=22,END=YES
00150 $SYSLOG  TERMINAL  DEVICE=4978,ADDRESS=04,HDCOPY=$SYSRTR
00160 $SYSLOGA TERMINAL  DEVICE=TTY,ADDRESS=00,CRODELAY=4,PAGSIZE=24,      C
00170          BOTM=23,SCREEN=YES
00180 $SYSLOGB TERMINAL  DEVICE=ACCA,ADDRESS=08,MODE=31013,ADAPTER=SINGLE,  C
00190          BITRATE=1200,RANGE=HIGH
00200 $SYSLOGC TERMINAL  DEVICE=ACCA,ADDRESS=60,MODE=31013,ADAPTER=FOUR,  C
00210          BITRATE=1200,RANGE=HIGH
00220 $SYSLOGD TERMINAL  DEVICE=ACCA,ADDRESS=68,MODE=31013,CODTYPE=ASCII,  C
00230          ATTN=1368,ADAPTER=FOUR,LF=0A,CR=0D,PF1=1361,      C
00240          BITRATE=1200,RANGE=HIGH
00250 $SYSRTR  TERMINAL  DEVICE=4974,ADDRESS=01,END=YES
00260 $SYSCCM  CSECT
00270          QCB
00280          QCB
00290          ECB
00300          ECB
00310          ENTRY   $EDXPTCH
00320 $EDXPTCH DATA   12BF'0'          SYSTEM PATCH AREA
00330          END

```

Figure 15-4. Contents of \$EDXDEF

The configuration statements shown match the hardware devices listed in Figure 15-3. You must edit this file to reflect your requirements (Figure 15-2). The 4963 DISK statement and \$SYSLOG TERMINAL statement must be modified. The 4966 DISK statement, the \$SYSLOGC and \$SYSLOGD TERMINAL statements for the 4978 at address 06 and a TIMER statement must be added to complete the new correct configuration. The configuration statements are now discussed in more detail.

## SYSTEM Statement

The SYSTEM statement (statement 110 in Figure 15-4) defines a 64K system (STORAGE=64), with a maximum of 10 programs executing concurrently (MAXPROG=10). Now, assume that the system this supervisor is being generated for has 128K of storage.

When a system has storage greater than 64K, multiple partitions must be defined, because of the way the software utilizes the hardware feature that addresses storage above 64K. Each partition defined is a separate relocatable program area, just as the space between the end of the supervisor and the end of storage is a relocatable area in systems with 64K or less.

The STORAGE= operand in the SYSTEM statement must be changed to STORAGE=128. Up to 8 partitions may be defined, and for this example, assume that 3 partitions are desired. The MAXPROG= operand will now be changed to MAXPROG=(10,10,10), with each sublist element in the operand list corresponding to the maximum number of programs allowed to execute in partition 1, partition 2, and partition 3, respectively. 10 programs in concurrent execution in any one partition is enough to exceed most application requirements, but this can be coded to meet your own application needs. (*Note:* All partitions do not have to have the same MAXPROG= value; MAXPROG=(6,3,10), for example, is valid.)

When using multiple partitions, a third operand, PARTS= must be coded. PARTS= is used to specify the size of each partition. Partitions can be up to 64K in size with the exception of Partition 1 which is restricted to 64K minus the size of the supervisor.

Partitions are defined in increments of 2K blocks (2048 bytes each). The first 64K of storage is represented by 32 such 2K blocks. If we estimate our supervisor to be 40K, we have 88K or 44-2K blocks of storage available for partitions. The largest size for partition one would be 24K (64 - 40K).

Let's assume in our system we desire partitions of 16, 32, and 40K. Using the text editor, the SYSTEM statement would be modified to

```
SYSTEM STORAGE=128,MAX PROG=(10,10,10),PARTS=(8,16,20)
```

## TIMER Statement

One of the devices to be supported by the new supervisor is Timers. The starter supervisor has no TIMER statement, so one must be added using the INSERT function of the Text Editor.

Although both timers will be supported, only one TIMER statement is entered. The system knows that the two timers have contiguous addresses, so a single TIMER statement specifying the address of the first timer is all that is required.

*Note:* For 4952 processors, the timer is part of the processor, not a feature and no TIMER system configuration statement is used.

## DISK Statements

The DISK configuration statement is used to define disk and diskette devices to the system. The configuration file (Figure 15-7) shows 3 DISK statements (4963 Model 23, 4964 Diskette and 4966 Diskette Magazine). Based on our hardware configuration, we must change the Disk to 4962-3 and delete the 4966 Diskette Magazine DISK statement.

To run disk/diskette devices, the system generates a system disk task which it attaches to perform disk or diskette I/O. As with any other task, the system disk task is not reentrant; it may only be attached by one user at a time. When multiple direct access devices are supported, where I/O requests to the high data rate disk could be suspended, waiting for the disk task to complete a request for one of the relatively slower diskette devices.

By coding TASK=YES in the DISK statements defining the diskette devices, a separate task is generated for each device.

After editing the configuration file, the DISK statements would be:

```
DISK  DEVICE = 4962-3,ADDRESS=03
DISK  DEVICE = 4964,ADDRESS=02,TASK=YES,END=YES
```

## TERMINAL Statement

Figure 15-5 below shows a list of the TERMINAL system configuration statements for the supplied supervisor.

```
00150 $SYSLOG  TERMINAL  DEVICE=4979,ADDRESS=04,HDCOPY=$SYSPRTR
00160 $SYSLOGA TERMINAL  DEVICE=TTY,ADDRESS=00,CRDELAY=+,PAGESIZE=24,
00170          BOTM=23,SCREEN=YES
00180 $SYSLOGB TERMINAL  DEVICE=ACCA,ADDRESS=08,MODE=3101B,ADAPTER=SINGLE,
00190          BITRATE=1200,RANGE=HIGH
00200 $SYSLOGC TERMINAL  DEVICE=ACCA,ADDRESS=60,MODE=3101B,ADAPTER=FOUR,
00210          BITRATE=1200,RANGE=HIGH
00220 $SYSLOGD TERMINAL  DEVICE=ACCA,ADDRESS=68,MODE=3101B,CODTYPE=ASCII,
00230          ATTN=1B68,ADAPTER=FOUR,LF=0A,CR=0D,PF1=1B61,
00240          BITRATE=1200,RANGE=HIGH
00250 $SYSPRTR  TERMINAL  DEVICE=4974,ADDRESS=01,END=YES
```

Figure 15-5. Starter TERMINAL statements

In a multiple partition system, terminals are assigned to partitions. When a terminal is assigned to a partition, operator commands invoked from that terminal are directed to the assigned partition. See the "OPERATOR COMMANDS" topic in "Section 14. System Utilities" for a discussion on how terminal/partition assignments may be changed online. For this SYSGEN, \$SYSLOG (statement 150) will be assigned to partition 1, and the TTY device (statements 160 and 170) will be assigned to partition 2. In statement 170 (the continuation of statement 160), the SCREEN= operand is coded as SCREEN=YES. This indicates that the supplied supervisor assumes that the TTY is an electronic display screen device.

SCREEN=YES causes a pause after every 24 lines of output, so that the data on the screen can be read by the operator. To display the next 24 lines, the operator must press the ENTER key.

Assume the TTY device on this system is not an electronic display screen device, but is a hardcopy TTY with continuous forms. The pause after every 24 lines is not required, and is in fact an annoyance, so SCREEN=YES should be changed to SCREEN=NO.

Using the delete function of the Text Editor, the \$SYSLOG and \$SYSLOGD statements can be removed from the file.

Next, the 4978 Display at address 06 and the 4973 Printer, neither of which is supported by the supplied supervisor, are added to the TERMINAL definitions. The 4978 is named DSPY1, and the 4973 LINEPRTR. The names used are not predefined; you must name the devices anything you wish.

After editing the configuration file, the TERMINAL statements would be:

```
00150 $SYSLOG   TERMINAL   DEVICE=4979,ADDRESS=04,HDCOPY=$SYSPRTR,PART=2
00151 DSPLY1    TERMINAL   DEVICE=4979,ADDRESS=06,HDCOPY=$SYSPRTR,PART=3
00160 $SYSLOGA  TERMINAL   DEVICE=TTY,ADDRESS=00,CRDELAY=4,PAGSIZE=24,      C
00170          9OTM=23,SCREEN=YES
00180 $SYSLOGB  TERMINAL   DEVICE=ACCA,ADDRESS=08,MODE=3101B,ADAPTER=SINGLE,  C
00190          3ITRATE=9600,RANGE=HIGH
00250 LINEPRTR  TERMINAL   DEVICE=4973,ADDRESS=21
00251 $SYSPRTR  TERMINAL   DEVICE=4974,ADDRESS=01,END=YES
```

Figure 15-6. Modified TERMINAL statements

### System Communications Area

The system communications area is the area known by the system global name \$SYSCOM. It is used for communication and synchronization between programs. The supplied supervisor already has a 128 word area called \$EDXPTCH, which can be used as part of \$SYSCOM.

For this example, \$SYSCOM will consist of two QCBs and two ECBs, plus the already existing 128 word area, \$EDXPTCH.

This completes the modifications to the system configuration source file. Figure 15-7 provides a listing of the changes made for the hardware specified earlier.

```

00010 $EDXDEF  CSECT
00020          DATA  F'0'
00030 *
00040 * EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
00050 *
00060 * THE FOLLOWING DEFINES THE STARTER SUPERVISOR AS SHIPPED ON THE
00070 * DISKETTE LABELED XS3001. FOR COMPLETE DESCRIPTIONS OF THESE
00080 * STATEMENTS OR ANY OTHER SYSTEM DEFINITION STATEMENTS, REFER TO
00090 * THE EDX VERSION 3 SYSTEM GUIDE: SC34-1702
00100 *
00110          SYSTEM STORAGE=64,MAXPROG=10,PARTS=32
00120          DISK  DEVICE=4963-23,ADDRESS=48
00130          DISK  DEVICE=4964,ADDRESS=02
00140          DISK  DEVICE=4966,ADDRESS=22,END=YES
00150 $SYSLOG  TERMINAL  DEVICE=4978,ADDRESS=04,HDCOPY=$SYSPRTR
00160 $SYSLOGA TERMINAL  DEVICE=TTY,ADDRESS=00,CRDELAY=4,PAGSIZE=24,      C
00170          BOTM=23,SCREEN=YES
00180 $SYSLOGB TERMINAL  DEVICE=ACCA,ADDRESS=08,MODE=3101B,ADAPTER=SINGLE,  C
00190          BITRATE=1200,RANGE=HIGH
00200 $SYSLOGC TERMINAL  DEVICE=ACCA,ADDRESS=60,MODE=3101B,ADAPTER=FOUR,  C
00210          BITRATE=1200,RANGE=HIGH
00220 $SYSLOGD TERMINAL  DEVICE=ACCA,ADDRESS=68,MODE=3101B,CODTYPE=ASCII,  C
00230          ATTN=1B68,ADAPTER=FOUR,LF=0A,CR=0D,PF1=1B61,          C
00240          BITRATE=1200,RANGE=HIGH
00250 $SYSPRTR TERMINAL  DEVICE=4974,ADDRESS=01,END=YES
00260 $SYSCDM  CSECT
00270          QCB
00280          QCB
00290          ECB
00300          FCB
00310          ENTRY  $EDXPTCH
00320 $EDXPTCH DATA  12BF'0'          SYSTEM PATCH AREA
00330          END

```

Figure 15-7. Starter configuration

The completed, edited file must be stored in a data set \$EDXDEFS on volume EDX002.

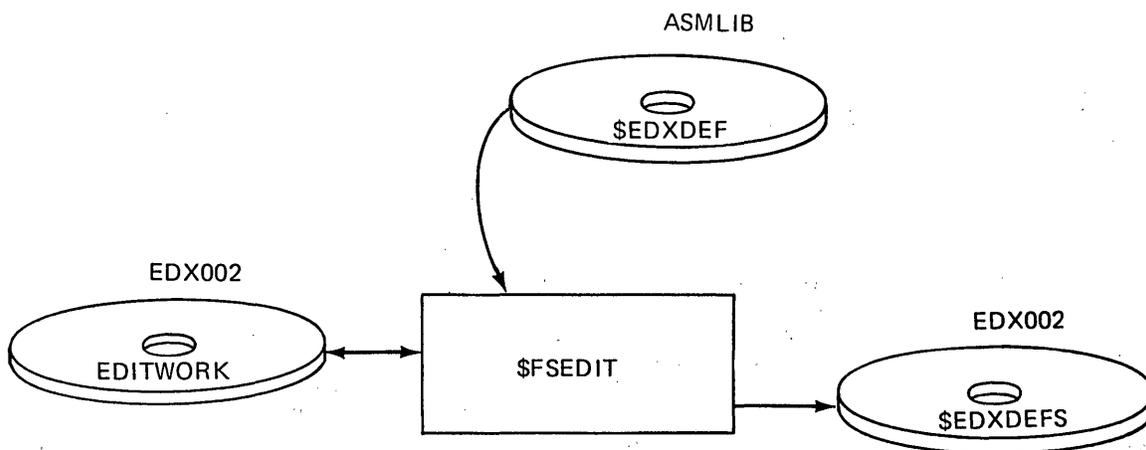


Figure 15-8. Configuration file update

## SELECT SUPERVISOR SUPPORT MODULES

The next step is to choose the supervisor support object modules required to support your configuration. These object modules are specified in link editor INCLUDE statements, which reside in a link edit control statement file.

As with the system configuration statements, you do not have to enter each INCLUDE record you need. Data set \$LNKCNTL on volume ASMLIB contains all possible supervisor INCLUDE statements. You must choose those required for your configuration. Figure 15-9 is a listing of the \$LINKCNTL data set that shows the modules that were included in the generation of the starter supervisor. The statements that have an asterisk in Column 1 are understood by the link editor to be comments rather than a control record.

```

00010 *
00020 * EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
00030 *
00090 OUTPUT SUPVLINK,EDX002 ENTRY=$START
00140 INCLUDE EDXSYS,XS3002 *0* SYSTEM TABLES AND WORK AREAS
00150 INCLUDE ASM0BJ,EDX002 *0* OUTPUT FROM USER SYSTEM GENERATION
00160 * INCLUDE EDXSVCX,XS3002 *0,K* TASK SUPERVISOR (XL)
00170 INCLUDE EDXSVCXU,XS3002 *0,L* TASK SUPERVISOR (UN-XL)
00180 * INCLUDE $DEBUGUC,XS3002 *6* RESIDENT $DEBUG SUPPORT
00190 INCLUDE EDXALU,XS3002 *0* EDL INSTRUCTION EMULATOR
00200 INCLUDE EDXSTART,XS3002 *0* INITIALIZATION & ERROR HANDLER
00260 INCLUDE DISKID,XS3002 *M* BASIC DISK(ETTE) SUPPORT
00270 INCLUDE D49624,XS3002 *M* 4962/4964 DISK(ETTE) SUPPORT
00280 INCLUDE D4963A,XS3002 *M* 4963 SUBSYSTEM SUPPORT
00290 INCLUDE D4966A,XS3002 *M* 4966 MAGAZINE SUPPORT
00350 * INCLUDE D4969A,XS3002 *M* BASIC TAPE SUPPORT
00410 * INCLUDE EDXTIO,XS3002 *1,K* BASIC TERMINAL SUPPORT (XL)
00420 INCLUDE EDXTIOU,XS3002 *1,L* BASIC TERMINAL SUPPORT (UN-XL)
00430 * INCLUDE EDXTERM,XS3002 *1,K* ENQ/DEQ & TERMINAL QUEUEING (XL)
00440 INCLUDE EDXTRMU,XS3002 *1,L* ENQ/DEQ & TERMINAL QUEUEING (UN-XL)
00450 * INCLUDE IOS4979,XS3002 *M,K* 4978/4979 DISPLAY SUPPORT (XL)
00460 INCLUDE IOS4979U,XS3002 *M,L* 4978/4979 DISPLAY SUPPORT (UN-XL)
00470 * INCLUDE IOS4974,XS3002 *M,K* 4973/4974 PRINTER SUPPORT (XL)
00480 INCLUDE IOS4974U,XS3002 *M,L* 4973/4974 PRINTER SUPPORT (UN-XL)
00490 INCLUDE IOSTERM,XS3002 *2* REQUIRED FOR TTY, ACCA, 4013 & 2741
00500 INCLUDE IOSTTY,XS3002 *M* ASP 33/35 TELETYPEWRITER SUPPORT
00510 INCLUDE IOSACCA,XS3002 *3* ASCII ACCA TERMINAL SUPPORT
00520 INCLUDE IOS3101,XS3002 *M,0* 3101 BLOCK MODE SUPPORT
00530 * INCLUDE IOSSIS1,XS3002 *M* SERIES/1 - SERIES/1 SUPPORT
00540 * INCLUDE IOSGP1B,XS3002 *M* GPIB SUPPORT
00550 * INCLUDE IOS4013,XS3002 *M* DIGITAL I/O TERMINAL SUPPORT
00560 * INCLUDE IOS2741,XS3002 *M* 2741 TERMINAL SUPPORT
00570 * INCLUDE IOSVIRT,XS3002 *M,N* VIRTUAL TERMINAL SUPPORT
00630 INCLUDE TRASCII,XS3002 *4,P* TELETYPEWRITER TRANSLATION
00640 INCLUDE TREBASC,XS3002 *3,P* MIRROR IMAGE ASCII TRANSLATION
00650 * INCLUDE TREBCD,XS3002 *5* 2741 EBCD TRANSLATION
00660 * INCLUDE TRCRSP,XS3002 *5* 2741 CORRESPONDENCE TRANSLATION
00720 * INCLUDE IGSPPOOL,XS3002 *M* SPOOLING SUPPORT
00780 * INCLUDE EDXTIMER,XS3002 *6* 4953/4955 TIMER (7840) SUPPORT
00790 * INCLUDE EDXTIMR2,XS3002 *6* 4952 TIMER SUPPORT
00850 * INCLUDE BSCAM,XS3002 *7,K* BISYNC COMM. ACCESS SUPPORT (XL)
00860 * INCLUDE BSCAMU,XS3002 *7,L* BISYNC COMM. ACCESS SUPPORT (UN-XL)
00870 * INCLUDE TPCOM,XS3002 *8* HJST COMMUNICATION SUPPORT

```

Figure 15-9. Starter Link file (1 of 2)

```

00930 * INCLUDE SBCOM,XS3002 *9* BASIC SENSOR I/O SUPPORT
00940 * INCLUDE IOLOADER,XS3002 *9,K* SENSOR I/O DEVICE OPEN (XL)
00950 * INCLUDE IOLOADERU,XS3002 *9,L* SENSOR I/O DEVICE OPEN (UN-XL)
00960 * INCLUDE SBAI,XS3002 *M* ANALOG INPUT SUPPORT
00970 * INCLUDE SBAO,XS3002 *M* ANALOG OUTPUT SUPPORT
00980 * INCLUDE SBDIO,XS3002 *M* DIGITAL INPUT/OUTPUT SUPPORT
00990 * INCLUDE SBPI,XS3002 *M* PROCESS INTERRUPT SUPPORT
01050 * INCLUDE IOSEXIO,XS3002 *M* EXIO DEVICE CONTROL SUPPORT
01110 INCLUDE SYSLOG,XS3002 *A* I/O ERROR LOGGING
01120 * INCLUDE NOSYSLOG,XS3002 *A* NO I/O ERROR LOGGING
01130 INCLUDE CIRCBUFF,XS3002 *B* PROGRAM/MACHINE CHECK LOGGING
01190 * INCLUDE RLOADER,XS3002 *C,K* RELOCATING PROGRAM LOADER (XL)
01200 INCLUDE RLOADERU,XS3002 *C,L* RELOCATING PROGRAM LOADER (UN-XL)
01210 * INCLUDE EDXFLOAT,XS3002 *D* FLOATING POINT ARITHMETIC
01220 INCLUDE NOFLOAT,XS3002 *D* FOR SYSTEMS WITHOUT FLOATING POINT
01230 * INCLUDE EBFLCVT,XS3002 *E* EBCDIC/FLOATING POINT CONV.
01240 INCLUDE QUEUFIO,XS3002 *F* QUEUE PROCESSING SUPPORT
01300 INCLUDE EDXINIT,XS3002 *H* SUPERVISOR INITIALIZATION
01310 INCLUDE DISKINIT,XS3002 *M* DISK(ETTE) INITIALIZATION
01320 * INCLUDE TAPEINIT,XS3002 *M* TAPE INITIALIZATION
01330 INCLUDE LOADINIT,XS3002 *C* PROGRAM LOADER INITIALIZATION
01340 INCLUDE RW4963ID,XS3002 *M* 4963 FIXED HEAD REFRESH SUPPORT
01350 INCLUDE TERMINIT,XS3002 *1* TERMINAL INITIALIZATION
01360 INCLUDE INIT4978,XS3002 *M* 4978 DISPLAY INITIALIZATION
01370 * INCLUDE INIT4013,XS3002 *M* DIGITAL I/O TERMINAL INITIALIZATION
01380 INCLUDE $ACCLARAM,XS3002 *3* ACCA MULTI-LINE ADAPTER RAM LOAD
01390 * INCLUDE BSCINIT,XS3002 *7* BISYNC (BSCAM) INITIALIZATION
01400 * INCLUDE TPINIT,XS3002 *3* HCF (TPCOM) INITIALIZATION
01410 * INCLUDE TIMRINIT,XS3002 *6* 4953/4955 TIMER INITIALIZATION
01420 * INCLUDE CLOGINIT,XS3002 *6* 4952 TIMER INITIALIZATION
01430 * INCLUDE SBI0INIT,XS3002 *M* SENSOR I/O INITIALIZATION
01440 * INCLUDE EXIOINIT,XS3002 *M* EXIO INITIALIZATION
01450 * INCLUDE SIS1INIT,XS3002 *M,Q* SIS1 INITIALIZATION
02060 END

```

Figure 15-9. Starter Link file (2 of 2)

Instead of deleting undesired INCLUDE statements, it is preferable to insert an asterisk in column 1. The asterisk causes the link editor to treat the statement as a comment statement rather than a control record. This gives you a record of what support you have decided to leave out, which can be helpful if problems develop with the generated supervisor.

The support available for some system functions is provided in two versions—untranslated or translated— specified on the INCLUDE statement comments as UN-XL and XL respectively. The untranslated modules support systems with a memory of 64K or less while translated modules support systems with greater than 64K of memory.

The completed INCLUDE file is shown in Figure 15-10. Those statements with asterisks in column 1 are for features that are not desired or for I/O devices not installed.

```

00020 * EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
00030 *
00080 OUTPUT SUPVLINK,EDX002 ENTRY=$START
00140 INCLUDE EDXSYS,XS3002 *0* SYSTEM TABLES AND WORK AREAS
00150 INCLUDE ASMOBJ,EDX002 *0* OUTPUT FROM USER SYSTEM GENERATION
00160 INCLUDE EDXSVCX,XS3002 *0,K* TASK SUPERVISOR (XL)
00170 * INCLUDE EDXSVCXU,XS3002 *0,L* TASK SUPERVISOR (UN-XL)
00180 INCLUDE $DEBUGUC,XS3002 *G* RESIDENT $DEBUG SUPPORT
00190 INCLUDE EDXALU,XS3002 *0* EDL INSTRUCTION EMULATOR
00200 INCLUDE EDXSTART,XS3002 *0* INITIALIZATION & ERROR HANDLER
00260 INCLUDE DISKIO,XS3002 *M* BASIC DISK(ETTE) SUPPORT
00270 INCLUDE D49624,XS3002 *M* 4962/4964 DISK(ETTE) SUPPORT
00280 * INCLUDE D4963A,XS3002 *M* 4963 SUBSYSTEM SUPPORT
00290 * INCLUDE D4966A,XS3002 *M* 4966 MAGAZINE SUPPORT
00350 * INCLUDE D4969A,XS3002 *M* BASIC TAPE SUPPORT
00410 INCLUDE EDXTIO,XS3002 *1,K* BASIC TERMINAL SUPPORT (XL)
00420 * INCLUDE EDXTIOU,XS3002 *1,L* BASIC TERMINAL SUPPORT (UN-XL)
00430 INCLUDE EDXTERMQ,XS3002 *1,K* ENQT/DEQT & TERMINAL QUEUEING (XL)
00440 * INCLUDE EDXTRMQU,XS3002 *1,L* ENQT/DEQT & TERMINAL QUEUEING (UN-XL)
00450 INCLUDE IOS4979,XS3002 *M,K* 4978/4979 DISPLAY SUPPORT (XL)
00460 * INCLUDE IOS4979U,XS3002 *M,L* 4978/4979 DISPLAY SUPPORT (UN-XL)
00470 INCLUDE IOS4974,XS3002 *M,K* 4973/4974 PRINTER SUPPORT (XL)
00480 * INCLUDE IOS4974U,XS3002 *M,L* 4973/4974 PRINTER SUPPORT (UN-XL)
00490 INCLUDE IOSTERM,XS3002 *2* REQUIRED FOR TTY, ACCA, 4013 & 2741
00500 INCLUDE IOSTTY,XS3002 *M* ASR 33/35 TELETYPEWRITER SUPPORT
00510 INCLUDE IOSACCA,XS3002 *3* ASCII ACCA TERMINAL SUPPORT
00520 INCLUDE IOS3101,XS3002 *M,0* 3101 BLOCK MODE SUPPORT
00530 * INCLUDE IOSSIS1,XS3002 *M* SERIES/1 - SERIES/1 SUPPORT
00540 * INCLUDE IOSGPB,XS3002 *M* GPIB SUPPORT
00550 * INCLUDE IOS4013,XS3002 *M* DIGITAL I/O TERMINAL SUPPORT
00560 * INCLUDE IOS2741,XS3002 *M* 2741 TERMINAL SUPPORT
00570 * INCLUDE IOSVIRT,XS3002 *M,N* VIRTUAL TERMINAL SUPPORT
00630 INCLUDE TRASCII,XS3002 *4,P* TELETYPEWRITER TRANSLATION
00640 INCLUDE TREBASC,XS3002 *3,P* MIRROR IMAGE ASCII TRANSLATION
00650 * INCLUDE TREBCD,XS3002 *5* 2741 EBCD TRANSLATION
00660 * INCLUDE TRCRSP,XS3002 *5* 2741 CORRESPONDENCE TRANSLATION
00720 INCLUDE IOSPOOL,XS3002 *M* SPOOLING SUPPORT
00780 INCLUDE EDXTIMER,XS3002 *6* 4953/4955 TIMER (7840) SUPPORT
00790 * INCLUDE EDXTIMR2,XS3002 *6* 4952 TIMER SUPPORT
00850 * INCLUDE BSCAM,XS3002 *7,K* BISYNC COMM. ACCESS SUPPORT (XL)
00860 * INCLUDE BSCAMU,XS3002 *7,L* BISYNC COMM. ACCESS SUPPORT (UN-XL)
00930 * INCLUDE SBCUM,XS3002 *9* BASIC SENSOR I/O SUPPORT
00940 * INCLUDE IOLOADER,XS3002 *9,K* SENSOR I/O DEVICE OPEN (XL)
00950 * INCLUDE IOLOADERU,XS3002 *9,L* SENSOR I/O DEVICE OPEN (UN-XL)
00960 * INCLUDE SBAI,XS3002 *M* ANALOG INPUT SUPPORT
00970 * INCLUDE SBAO,XS3002 *M* ANALOG OUTPUT SUPPORT
00980 * INCLUDE SBDIDO,XS3002 *M* DIGITAL INPUT/OUTPUT SUPPORT
00990 * INCLUDE SBPI,XS3002 *M* PROCESS INTERRUPT SUPPORT
01050 * INCLUDE IOSEXID,XS3002 *M* EXIO DEVICE CONTROL SUPPORT
01110 INCLUDE SYSLOG,XS3002 *A* I/O ERROR LOGGING
01120 * INCLUDE NOSYSLOG,XS3002 *A* NO I/O ERROR LOGGING
01130 INCLUDE CIRCBUFF,XS3002 *B* PROGRAM/MACHINE CHECK LOGGING
01190 INCLUDE RLOADER,XS3002 *C,K* RELOCATING PROGRAM LOADER (XL)
01200 * INCLUDE RLOADERU,XS3002 *C,L* RELOCATING PROGRAM LOADER (UN-XL)
01210 INCLUDE EOXFLOAT,XS3002 *D* FLOATING POINT ARITHMETIC
01220 * INCLUDE NOFLOAT,XS3002 *D* FOR SYSTEMS WITHOUT FLOATING POINT

```

Figure 15-10. Updated Link file (1 of 2)

```

01230  INCLUDE  EBFLCVT,XS3002  *E*  EBCDIC/FLOATING POINT CONV.
01240  INCLUDE  QUEUEID,XS3002  *F*  QUEUE PROCESSING SUPPORT
01300  INCLUDE  EDXINIT,XS3002  *H*  SUPERVISOR INITIALIZATION
01310  INCLUDE  DISKINIT,XS3002  *M*  DISK(ETTE) INITIALIZATION
01320 * INCLUDE  TAPEINIT,XS3002  *M*  TAPE INITIALIZATION
01330  INCLUDE  LOADINIT,XS3002  *C*  PROGRAM LOADER INITIALIZATION
01340 * INCLUDE  RW4963ID,XS3002  *M*  4963 FIXED HEAD REFRESH SUPPORT
01350  INCLUDE  TERMINIT,XS3002  *1*  TERMINAL INITIALIZATION
01360  INCLUDE  INIT4978,XS3002  *4*  4978 DISPLAY INITIALIZATION
01370 * INCLUDE  INIT4013,XS3002  *M*  DIGITAL I/O TERMINAL INITIALIZATION
01380 * INCLUDE  $ACCARAM,XS3002  *3*  ACCA MULTI-LINE ADAPTER RAM LOAD
01390 * INCLUDE  BSCINIT,XS3002  *7*  BCSYNC (BSCAM) INITIALIZATION
01400 * INCLUDE  TPINIT,XS3002  *8*  HCF (TPCOM) INITIALIZATION
01410  INCLUDE  TIMRINIT,XS3002  *6*  4953/4955 TIMER INITIALIZATION
01420 * INCLUDE  CLOKINIT,XS3002  *6*  4952 TIMER INITIALIZATION
01430 * INCLUDE  SBIUNIT,XS3002  *M*  SENSOR I/O INITIALIZATION
01440 * INCLUDE  EXIOINIT,XS3002  *M*  EXIO INITIALIZATION
01450 * INCLUDE  SISLINIT,XS3002  *M,Q*  SISL INITIALIZATION
02060  END

```

Figure 15-10. Updated Link file (2 of 2)

The completed file is now saved to the LINKCNTL data set on volume EDX002.

Figure 15-11 summarizes operations up to this point.

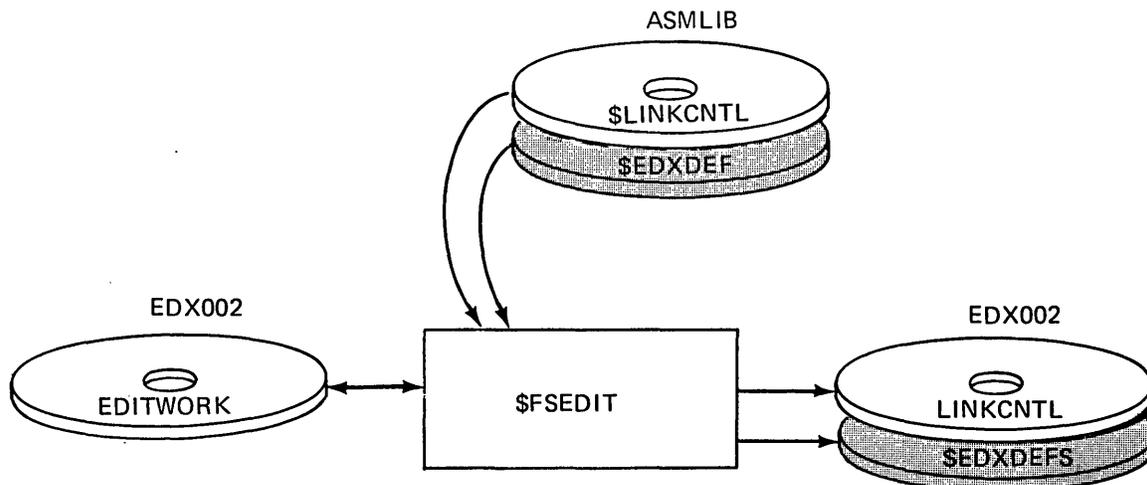


Figure 15-11. Link file update

## EDIT \$JOBUTIL PROCEDURE FILE

Now that \$EDXDEFS contains your system configuration statements, and LINKCNTL contains the edited INCLUDE file, you are ready to assemble the configuration statements, and link edit the resulting object module with the supervisor support object modules specified in LINKCNTL. The linked object module will then be formatted by the \$UPDATE utility to form an executable supervisor.

The assemble, link, and formatting steps will be performed under control of the job stream processing utility \$JOBUTIL. You could load the assembler \$EDXASM, provide the data set names required yourself, and do the assembly, then in turn do the same for \$LINK and \$UPDATE, but using \$JOBUTIL, all three steps may be accomplished with a single entry.

\$JOBUTIL operation is controlled by a procedure file of job control statements. For SYSGEN, a procedure file named \$SUPPREP is supplied on volume ASMLIB. Figure 15-12 presents a listing of that procedure file.

If, when you allocated data sets at the beginning of SYSGEN, you had used other than the names/volumes recommended, you would now have to edit this procedure file to reflect the names/volumes you used.

```
00010 *
00020 * EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
00030 *
00040 LOG      $SYSPRTR
00050 JOB      $SUPPREP
00060 REMARK    ** ENTER -GO- AFTER -XS3002- HAS BEEN VARIED ONLINE **
00070 REMARK    ** AND AFTER THE FOLLOWING MEMBERS **
00080 REMARK    ** HAVE BEEN ALLOCATED ON VOLUME EDX002: **
00090 REMARK    ** ASMWRK,ASMUBJ,LEWORK1,LEWORK2,SUPVLINK **
00100 REMARK    ** - SIZES AS PRESCRIBED IN PROGRAM DIRECTORY - **
00110 PAUSE
00120 PROGRAM  $EDXASM,ASMLIB      EDX ASSEMBLER PROGRAM
00130 NOMSG
00140 PARM
00150 DS      $EDXDEFS,EDX002      CONFIGURATION STATEMENTS DATA SET
00160 DS      ASMWRK,EDX002        ASSEMBLER WORK DATA SET
00170 DS      ASMUBJ,EDX002        OBJECT OUTPUT DATA SET
00180 EXEC
00190 JUMP    ENDJOB,GT,4
00200 PROGRAM  $LINK,EDX002        LINK EDITOR PROGRAM
00210 NOMSG
00220 PARM      $SYSPRTR
00230 DS      LINKCNTL,EDX002      LINK EDITOR CONTROL STATEMENTS
00240 DS      LEWORK1,EDX002      LINK EDITOR WORK DATA SET
00250 DS      LEWORK2,EDX002      LINK EDITOR WORK DATA SET
00260 EXEC
00270 JUMP    ENDJOB,GT,4
00280 PROGRAM  $UPDATE,EDX002      UPDATE (FORMAT) PROGRAM
00290 NOMSG
00300 PARM      $SYSPRTR SUPVLINK,EDX002 $EDXNUCT,EDX002 YES
00310 EXEC
00320 LABEL   ENDJOB
00330 F0J
```

Figure 15-12. \$JOBUTIL procedure

For example, if you had called the assembler work file ASMWRK1 instead of ASMWORK, you would have to change the name in the DS statement number 160.

All files allocated for this SYSGEN used the recommended names and volumes, so the editor work data set is saved in the data set SUPPREPS on EDX002. The editing portion of SYSGEN is complete, and is summarized in Figure 15-13.

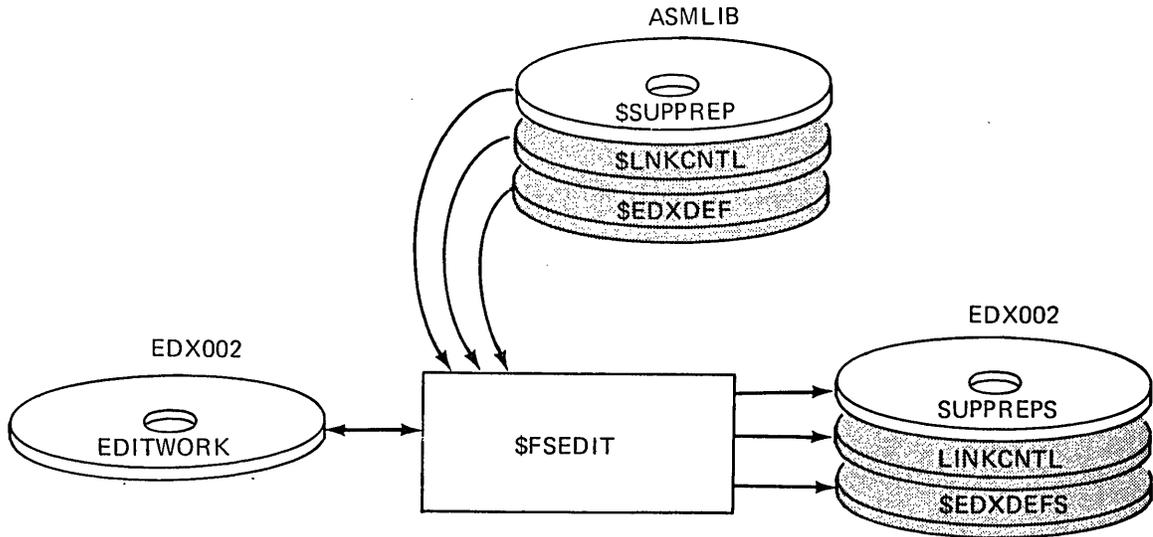


Figure 15-13. Procedure file update

*Note:* Because there were no changes required in the \$JOBUTIL procedure file, the transfer of \$SUPPREP on ASMLIB to SUPPREPS on EDX002 could have been accomplished using \$COPY or \$COPYUT1, rather than with the READ and WRITE text editor commands.

## ASSEMBLE/LINK/FORMAT

To assemble, link edit, and format the tailored supervisor, load \$JOBUTIL, and supply the name of your procedure file, as illustrated in Figure 15-14.

```
> $L $JOBUTIL
$JOBUTIL                4P, LP = 6000
ENTER PROCEDURE (NAME,VOLUME): SUPPREPS,EDX002

$JOBUTIL ENDED
```

Figure 15-14. \$JOBUTIL

The procedure file has specified \$SYSPRTR as the log device, so the first thing that happens is that the procedure file statements controlling the assembly operation print out on the system printer (see Appendix A, Figure A-1). \$JOBUTIL loads the assembler, \$EDXASM, which assembles your system configuration source file, \$EDXDEFS.

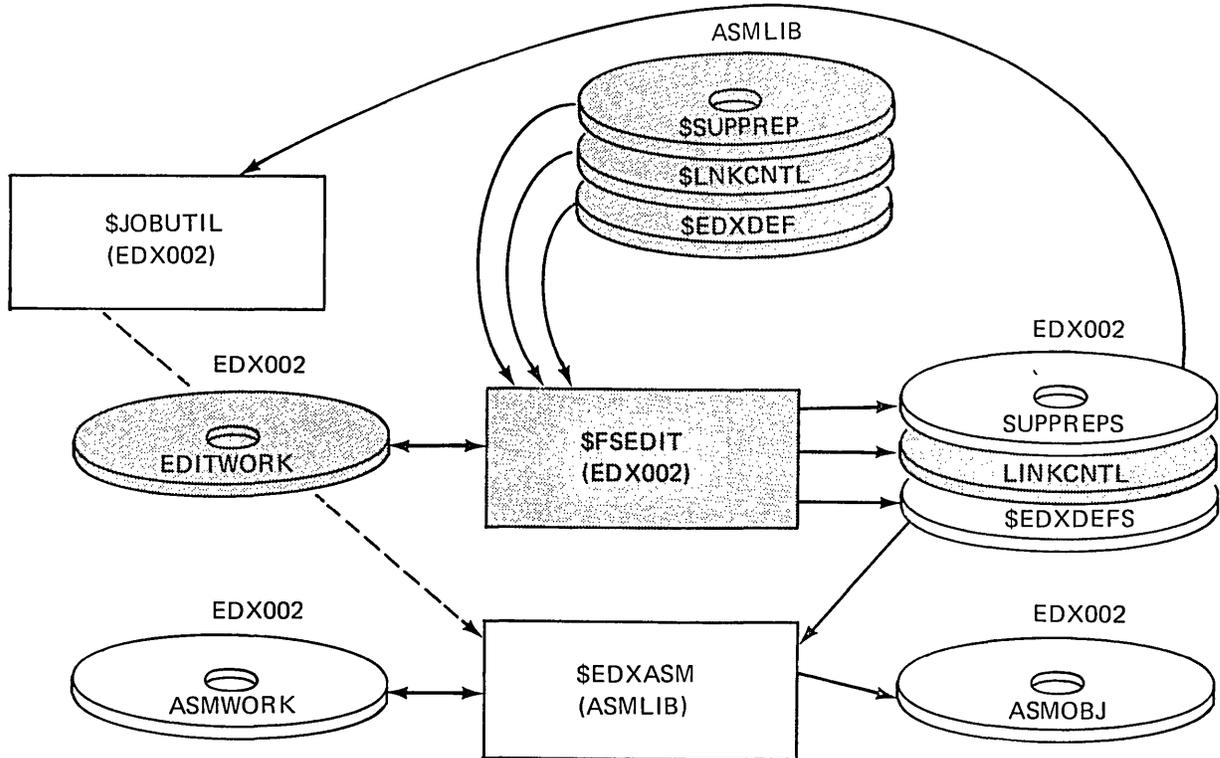


Figure 15-15. \$EDXASM

The resulting object module is stored in data file ASMOBJ on volume EDX002, which you created. The listing produced as a result of the assembly prints out on the system printer, preceded by assembler statistics (see Appendix A, Figure A-2).

Next, \$JOBUTIL loads the link editor, \$LINK. (Appendix A, Figure A-3.) Using the object module from the assembly (ASMOBJ) and the file of link control records (INCLUDE statements) you stored in LINKCNTL, the \$LINK program brings in the supervisor object modules specified in LINKCNTL and link edits them with the system control blocks generated by the assembly (ASMOBJ object module).

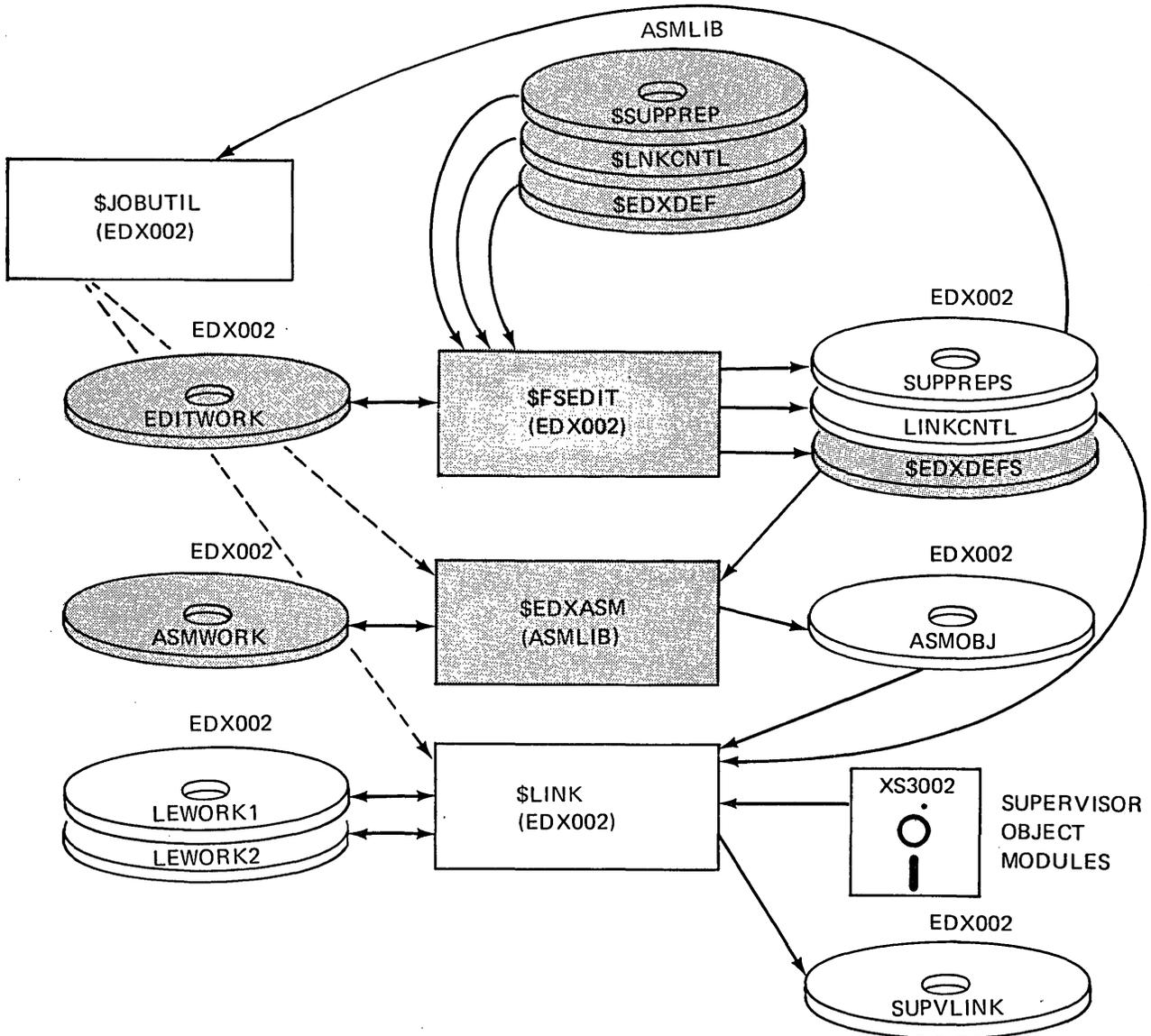


Figure 15-16. \$LINK

The data set SUPVLINK, which you allocated for link edit output, is used to store the resulting linked module. The link editor prints out the LINKCNTL file (Appendix A, Figure A-4) and any unresolved references resulting from the link edit on the system printer. There will be several unresolved weak external references (WXTRN) for supervisor support modules you did not want to include, but no unresolved EXTRN messages should appear.

\$JOBUTIL now loads \$UPDATE to format the linked supervisor into a loadable module (Appendix A, bottom of Figure A-4).

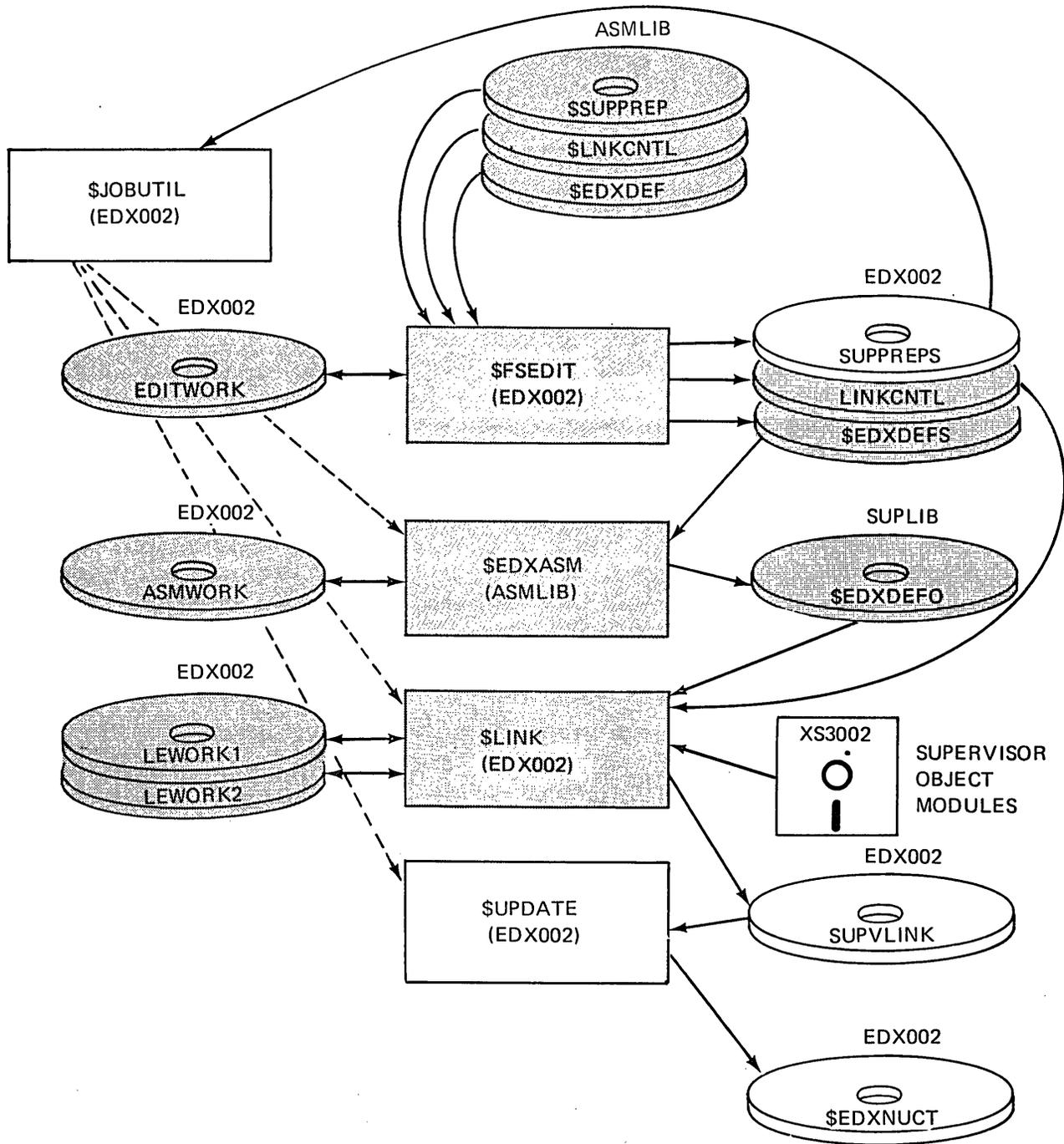


Figure 15-17. \$UPDATE

The formatted load module is placed in \$EDXNUCT, a supervisor data set allocated automatically by \$UPDATE. \$UPDATE ends (Appendix A, Figure A-5) and \$JOBUTIL completes.

## Designate Tailored Supervisor

Before you can test the new supervisor, it must be designated as the one to be loaded at IPL time. To do this you must invoke \$INITDSK and write a new IPL text record (command II) designating \$EDXNUCT on EDX002 as the new supervisor.

Figure 15-18 summarizes the tailored system generation process.

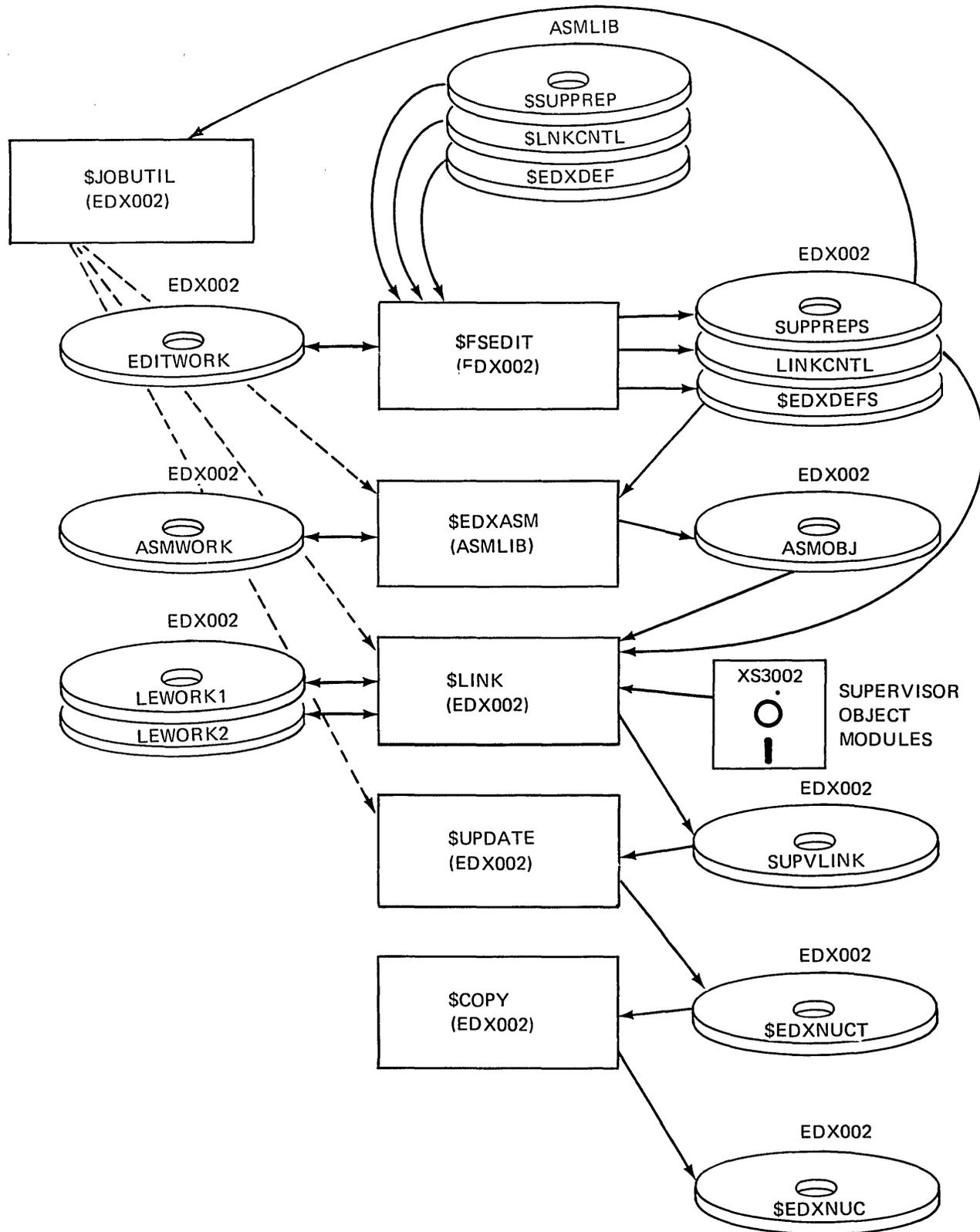


Figure 15-18. SYSGEN overview

## IPL Tailored Supervisor

When you IPL the tailored supervisor, the IPL message shown in Figure 15-19 is displayed.

```
*** EVENT DRIVEN EXECUTIVE ***  VER 3.0
IPL = SEDXNUCT,EDX002
STORAGE MAP
PART#  START  SIZE
  1    40448 16896
  2    57344 32768
  3    90112 40960
SET DATE AND TIME USING COMMAND ST
EDX INITIALIZATION COMPLETE
```

Figure 15-19. IPL message

The message on the \$SYSLOG device indicates that \$EDXNUCT is the supervisor that was loaded and that it is 40K bytes in size. Partition sizes are as shown. Users may now execute programs under the tailored system.

## Section 16. Program Preparation Using \$EDXASM

**OBJECTIVES:** After completing this section, the student should be able to;

1. Describe the steps required for application program preparation
2. Understand the operation of the online utilities/programs used for program preparation (5719-XX2)

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive System Guide (SC34-1702), "Program Preparation."

### PROGRAM PREPARATION OVERVIEW

The steps required to prepare an Event Driven Executive application program for execution are outlined in Figure 16-1.

#### STEP 1: CREATE SOURCE MODULE

Source program modules are created using either \$EDIT1N or \$FSEDIT, the text editing utilities.

#### STEP 2: ASSEMBLE SOURCE MODULE

\$EDXASM, the assembler program, produces object modules from source modules. An object module may be input to the link edit program \$LINK or, if no references to external modules are made, it may be input to the formatting utility \$UPDATE.

#### STEP 3: PRODUCE ASSEMBLY LISTING

This is a subfunction of the assembly, STEP 2. The listing can be suppressed entirely, or errors only printed. The listing may be directed to a device other than the system printer, if desired. The listing is produced by \$EDXLIST, a separate program loaded by \$EDXASM as required.

#### STEP 4: LINK EDIT OBJECT MODULES

The \$LINK program is used to combine object modules to form a complete program. This step is not required if the object module produced by an assembly is already a complete program in itself (no references to external modules included in the assembly).

#### STEP 5: FORMAT OBJECT MODULE

Program object modules produced by \$EDXASM or \$LINK are not in executable form. They must first be processed into relocatable load modules by the utility program \$UPDATE.

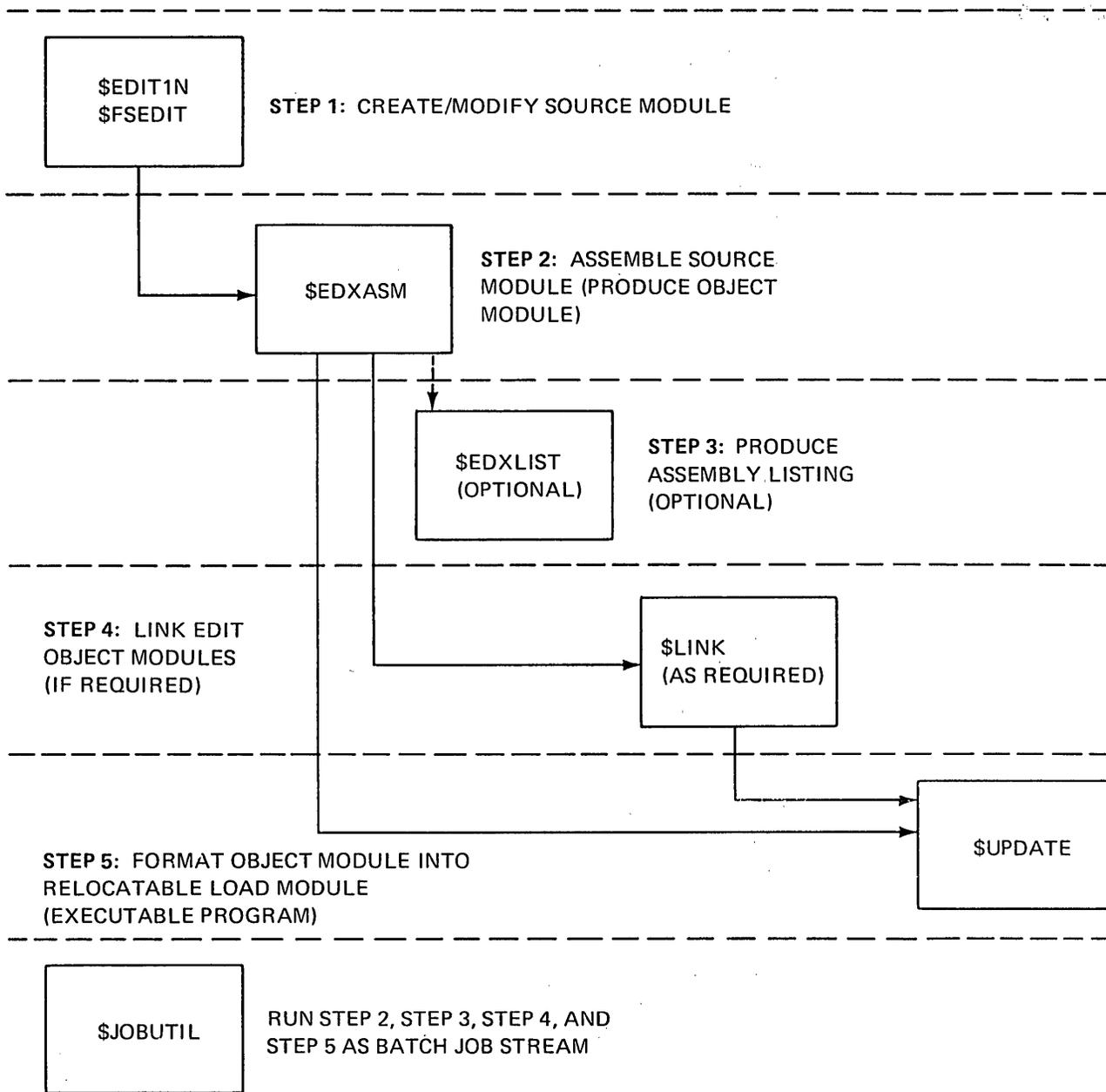


Figure 16-1. Program preparation overview

**\$JOBUTIL: BATCH JOB STREAM PROCESSOR**

At the bottom of Figure 16-1 is a reference to \$JOBUTIL, the batch job stream processor. This is a program preparation productivity aid which allows the assembly, link edit, and formatting steps to be run as a continuous sequence of job steps, without operator intervention.

In this section, the features and operating characteristics of each of the programs/utilities required for program preparation is discussed separately. Following the discussion is a comprehensive example, using each utility in preparing a program for execution.

## \$EDXASM

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$EDXASM."

\$EDXASM is the system program used for assembly of source programs written in the Event Driven Executive language. \$EDXASM, along with other program preparation programs, resides on volume ASMLIB.

*Data Set Requirements.* \$EDXASM is loaded using the "\$L" supervisor utility function. The operator will be prompted for required data set names, as shown in Figure 16-2.

```
> $L $EDXASM,ASMLIB  
  
SOURCE (NAME,VOLUME): SRCINPUT  
WORKFILE (NAME,VOLUME): WORKSET  
OBJECT (NAME,VOLUME): OBJOUT
```

Figure 16-2. \$EDXASM (1)

The SOURCE data set is the input source module to be assembled. The statements in this file are created using \$EDIT1N or \$FSEDIT.

For WORKFILE, enter the name of a data set to be used as an assembler work area. This file must already be allocated, and usually ranges between 100 and 500 records in size, with 250 about average.

The OBJECT data set is the preallocated data set in which the object module resulting from the assembly will be stored. This object module will be input either to \$LINK, if it is to be combined with other object modules, or to \$UPDATE, if it is a complete program (no references to external modules).

In Figure 16-2, all three data sets reside on the IPL volume, as no volume names are supplied. Were the data sets resident on other volumes, each data set name would be followed by the volume, separated by a comma.

The loader (\$L function) is a serially reusable resource. In Figure 16-2, the loader is enqueued, and therefore unavailable to other users and to the system, as soon as the ENTER key is pressed to enter the first line, \$L \$EDXASM,ASMLIB. It remains enqueued throughout the prompt/response sequence that follows, a length of time which may be considerable, depending on how familiar the operator is with the data set names requested, and how fast they can be entered.

```
> $L $EDXASM,ASMLIB SRCINPUT WORKSET OBJOUT
```

Figure 16-3. \$EDXASM (2)

Figure 16-3 illustrates an alternate way of entering the same load request. When the ENTER key is pressed, all required data set names are available on the same line, and enqueue time for the loader is greatly reduced. For \$EDXASM, and all other utilities accepting advance input, the advance input form should be used where possible. *Note:* Utilities accepting advance input have no way of "knowing" the purpose of a data set, other than by the position of the data set name on the advance input line. The data set names must be supplied on the advance input line in the same sequence as the utility would prompt for them were advance input not employed.

In addition to source, work, and object data sets, whose names must be supplied at load time, \$EDXASM also uses a language control data set. The language control data set supplied with the system is called \$EDXL and contains the assembler error messages and an "op code to processing module" specification for each of the standard Event Driven Executive instructions. If users wish to modify the instruction set or add error messages, \$EDXL may be changed, or a new language control data set produced (the language control data set is in source statement format, and can be modified using \$EDIT1N or \$FSEDIT).

\$EDXASM supports the copycode function, which allows source code residing in data sets to be included in an assembly by coding a COPY statement in the source program. The language control data set is used to define disk or diskette volumes containing copycode data sets to the assembler.

\$EDXL, the system-supplied language control data set, already contains \*COPYCOD statements which define disk volumes ASMLIB and EDX002 as volumes containing copycode data sets. If a user-written copycode data set resides on either of these volumes, no change to \$EDXL is required to use the COPY statement in a user source program assembly. However, if a user copycode data set resides on a volume other than ASMLIB or EDX002, \$EDIT1N or \$FSEDIT must be used to add a \*COPYCOD statement to \$EDXL which defines the new volume as one which may contain copycode data sets.

After \$EDXASM has been loaded the SELECT OPTIONS (?): prompt will appear. A "?" response will list the available options, as shown in Figure 16-4.

```
SELECT OPTIONS (?) : ?  
LIST      - SPECIFY LIST DEVICE  
NOLIST    - DO NOT PRINT LISTING  
ERRORS    - LIST ERRORS ONLY  
CONTROL   - SPECIFY CONTROL LANGUAGE  
END       - END OPTION SELECTION  
( 'ATTN - CA' TO CANCEL ASSEMBLY)
```

Figure 16-4. \$EDXASM (3)

- LIST** You can specify the name of the device that will be used for the assembly listing (name=label in **TERMINAL** system configuration statement). If the **LIST** option is not entered, the list device will default to **\$\$SYSPRTR**.
- NOLIST** This option suppresses the listing entirely, but assembly statistics will be displayed on the loading terminal.
- ERRORS** Only statements causing assembly errors, along with their error messages, will be listed. The operator will also be prompted for the name of the error list device.
- CONTROL** You can specify the name of your own language control data set. If it is not entered, this option defaults to **\$EDXL** on volume **ASMLIB**.
- END** Once any option is entered in response to the **SELECT OPTIONS (?)**: prompt, the operator will continue to be prompted until **END** is entered, or until the **ENTER** key is depressed with no entry. If no response is made to the first **SELECT OPTIONS (?)**: prompt (**ENTER** key with nothing entered), the assembly will start without **END**'s being entered, **\$EDXL** on **ASMLIB** will be used as the language control data set, and the full listing will appear on the system printer (**\$\$SYSPRTR**).

## **\$EDXLIST**

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$EDXLIST."

The assembly listing is produced by the assembly list processing program **\$EDXLIST**. Though usually run as part of the assembly process, **\$EDXLIST** may be loaded directly (**\$L**) and run after the assembly is finished, as long as the assembler work data set has not been disturbed (used in another assembly). See the reading assignment for operating instructions.

## **\$LINK**

**READING ASSIGNMENT:** IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$LINK—Linkage Editor."

**\$LINK** is used to combine two or more object modules into a single output object module. Input object modules may be produced by **\$EDXASM**, by the Series/1 macro assembler (**\$\$1ASM**), by the **PL/I**, **FORTRAN** or **COBOL** compilers, or by the Host Assembler. The output object module produced by **\$LINK** must be processed by **\$UPDATE** before it can be loaded and executed.

*Data Set Requirements.* When \$LINK is loaded, the operator is prompted for the names of three data sets. The first is the link control data set, which will contain control records specifying the object modules (names of object module data sets) that will be linked together. The other two data set names are the names of link edit work data sets, used as work areas during the linkedit process.

```
> $L $LINK
LINKCNTL (NAME, VOLUME): LINKCNTL
LEWORK1 (NAME, VOLUME): LINKWRK1
LEWORK2 (NAME, VOLUME): LINKWRK2
$LINK          76P,00:40:39, LP= 5F00

ENTER DEVICE NAME FOR PRINTED OUTPUT
$SYSPRTR
```

Figure 16-5. \$LINK (1)

See the reading assignment for recommended work data set sizes.

The link control data set (LINKCNTL) controls overall link edit operation. The control records are produced using \$EDIT1N or \$FSEDIT. The first control record in all LINKCNTL data sets is an OUTPUT statement, specifying the data set that will be used to store the output object module resulting from the link edit. This data set (as well as the work data sets) must be allocated before the link operation is attempted. In Figure 16-6, the output statement specifies data set LINKOUT on the IPL volume (if no volume is specified, default=IPL) as the output data set for the linked object module.

```
OUTPUT LINKOUT
INCLUDE ASMOUT1,EDX003
INCLUDE ASMOUT5
END
```

Figure 16-6. \$LINK (2)

The output object module will be produced by linking the input object module in ASMOUT1 on volume EDX003 with the object module in ASMOUT5 on the IPL volume, as specified by the two INCLUDE statements following the OUTPUT record. The first INCLUDE record must specify an object module that contains an initial task, produced by an assembly of a source module beginning with a PROGRAM statement with the MAIN= operand coded as (or defaulted to) MAIN=YES. Subsequent INCLUDE records cannot specify object modules containing initial tasks.

In addition to those object modules explicitly named in INCLUDE statements, \$LINK can also include object modules through the AUTOCALL option. Using the AUTO= operand of the OUTPUT control record, an autocall definition data set may be named. This data set contains the names (and volumes, if not IPL resident) of autocall object modules, along with their entry points.

```

OUTPUT LINKOUT AUTO=MYAUTO,EDX003
INCLUDE ASMOUTA
INCLUDE ASMOUTB
END
RENBR,EDX001      RENUM1      RENUM2
ABTERM            ABENT        **END

```

Figure 16-7. \$LINK (3)

In Figure 16-7, a reference to RENUM1, RENUM2, or ABENT from within object module ASMOUTA or ASMOUTB cannot be resolved by linking ASMOUTA with ASMOUTB. Because AUTO= is coded, \$LINK goes to the autocall data set MYAUTO, and tries to find the referenced name in the list of entry points specified in the autocall definition records. If a match is found, \$LINK will link the associated autocall object module with ASMOUTA and ASMOUTB.

The \*\*END in the last autocall definition record performs the same function for the autocall definition data set as does the END record for the link control data set.

In addition to the link control and work data set names, the operator is also prompted for the name (label of TERMINAL system configuration statement) of the terminal which is to receive the \$LINK output messages (see Figure 16-5). \$LINK prints out the link control statement file, and a map of the linked object module (see the reading assignment for an example).

## \$JOBUTIL

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference, Messages and Codes (SC34-1703), "\$JOBUTIL — Job Stream Processor."

\$JOBUTIL is the batch job stream processor utility. \$JOBUTIL uses a user-created (\$EDIT1N, \$FSEDIT) job processor procedure file to sequentially execute a series of programs. To illustrate basic \$JOBUTIL operation, a procedure file to invoke the online assembler, \$EDXASM will be created.

Procedure command statements are stored two statements per record, so a data set size of 15 or 20 records is usually adequate. For this discussion, assume a data set called MYPROC is allocated on the IPL volume.

Using \$EDIT1N or \$FSEDIT, the procedure command file can now be created. An asterisk in column 1 defines an internal comment command.

```

{
* $JOBUTIL / $EDXASM EXAMPLE
}

```

Figure 16-8. \$JOBUTIL (1)

The entire statement is treated as a comment, and may appear anywhere within the procedure command file. The internal comment statements are for procedure file documentation only; they are not printed out or displayed during \$JOBUTIL operation.

All the other procedure commands have a defined positional format. The commands must appear in character positions 1 through 8, starting in 1; operands in 10 through 17, starting in 10; and comments in 18 through 71.

```
}
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
{
```

Figure 16-9. \$JOBUTIL (2)

The LOG command controls the printing of \$JOBUTIL procedure commands, with LOG coded as shown in Figure 16-9, procedure commands will be displayed on the terminal used to load \$JOBUTIL, as they are read from the procedure file. Other operand options are either OFF, for no logging of procedure commands, or terminal name specifying the name of a terminal to which you wish the \$JOBUTIL procedure commands directed.

```
{
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
* 'REMARK' COMMAND - DISPLAYS MESSAGE
* ON LOADING TERMINAL
REMARK  OPERATOR MESSAGE
{
```

Figure 16-10. \$JOBUTIL (3)

The REMARK command will display on the terminal used to load \$JOBUTIL. REMARK commands may be placed anywhere within a procedure file. The JOB command, like the REMARK command, is optional. In Figure 16-11, the JOB command is the first command in the procedure data set, but could follow the LOG or the REMARK. The JOB command displays a "job started" message on the loading terminal, with the time and date. Both JOB and REMARK operate without regard to LOG (LOG OFF has no effect).

```

JOB      ASMPLE
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
* 'REMARK' COMMAND - DISPLAYS MESSAGE
* ON LOADING TERMINAL
REMARK   OPERATOR MESSAGE
* 'PROGRAM' COMMAND DEFINES THE PROGRAM
* TO BE LOADED
PROGRAM  $EDXASM,ASMLIB
{
}

```

Figure 16-11. \$JOBUTIL (4)

The PROGRAM command defines the program name/volume that \$JOBUTIL is to load (if the JOB command is used, it must appear before PROGRAM).

```

JOB      ASMPLE
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
* 'REMARK' COMMAND - DISPLAYS MESSAGE
* ON LOADING TERMINAL
REMARK   OPERATOR MESSAGE
* 'PROGRAM' COMMAND DEFINES THE PROGRAM
* TO BE LOADED
PROGRAM  $EDXASM,ASMLIB
* 'DS' COMMANDS DEFINE DATA SETS THE
* LOADED PROGRAM REQUIRES
DS       SCRMAT
DS       ASMWORK
DS       ASMOUT2
{
}

```

Figure 16-12. \$JOBUTIL (5)

“DS” commands define data sets to the program being loaded. Only one data set may be defined with each DS statement, and the definitions must appear in the same order as the responses to load-time data set definition prompts would be entered, were the program loaded using the “\$L” operator command.

Following the DS commands, any additional information required by the program being loaded is passed using the PARM command. In Figure 16-13, PARM is coded with no operand. This is equivalent to responding to the SELECT OPTIONS: prompt by pressing the ENTER key without entering an option, when \$EDXASM is loaded using \$L.

```

JOB      ASMPLE
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
* 'REMARK' COMMAND - DISPLAYS MESSAGE
* ON LOADING TERMINAL
REMARK   OPERATOR MESSAGE
* 'PROGRAM' COMMAND DEFINES THE PROGRAM
* TO BE LOADED
PROGRAM  $EDXASM,ASMLIB
* 'DS' COMMANDS DEFINE DATA SETS THE
* LOADED PROGRAM REQUIRES
DS       SCRMAT
DS       ASMWORK
DS       ASMOUT2
* 'PARM' COMMAND PASSES PARAMETERS TO
* THE LOADED PROGRAM
PARM
{

```

Figure 16-13. \$JOBUTIL (6)

The program to be loaded now has all the information required to load and execute. In Figure 16-14, the "EXEC" command issues the load request for the program defined in the preceding PROGRAM command.

```

JOB      ASMPLE
* $JOBUTIL / $EDXASM EXAMPLE
* 'LOG' COMMAND - $JOBUTIL LOG DEFINITION
LOG      ON
* 'REMARK' COMMAND - DISPLAYS MESSAGE
* ON LOADING TERMINAL
REMARK   OPERATOR MESSAGE
* 'PROGRAM' COMMAND DEFINES THE PROGRAM
* TO BE LOADED
PROGRAM  $EDXASM,ASMLIB
* 'DS' COMMANDS DEFINE DATA SETS THE
* LOADED PROGRAM REQUIRES
DS       SCRMAT
DS       ASMWORK
DS       ASMOUT2
* 'PARM' COMMAND PASSES PARAMETERS TO
* THE LOADED PROGRAM
PARM
* 'EXEC' COMMAND ISSUES LOAD REQUEST FOR
* THE PROGRAM
EXEC
* 'EOJ' ENDS THE PROCEDURE COMMAND FILE
EOJ

```

Figure 16-14. \$JOBUTIL (7)

The "EOJ" command following the EXEC indicates end of job, and terminates the job stream processor utility. If another job were to be run before ending this procedure, appropriate PROGRAM, DS, PARM and EXEC statements would precede the EOJ.

When the text editing session that created the procedure is complete, the procedure is stored in the data set MYPROC. The job can be run by loading \$JOBUTIL, and specifying procedure file MYPROC, as shown in Figure 16-15.

```
> $L $JOBUTIL
$JOBUTIL      3P,00:00:17, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): MYPROC
```

Figure 16-15. \$JOBUTIL (8)

In Figure 16-16, each of the procedure command statements in procedure file MYPROC (without the internal comments) is related to the equivalent operator responses for a \$L load of the assembler.

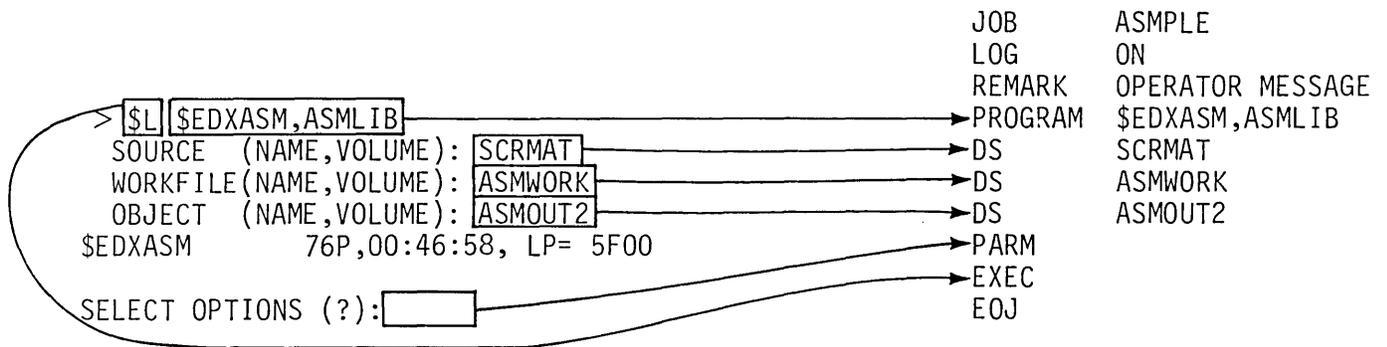


Figure 16-16. \$JOBUTIL (9)

Other \$JOBUTIL commands allow job steps to be skipped/executed based on the completion code returned from a previous step, the invoking of nested procedures in other procedure data sets, and the entering of procedure commands from the loading terminal. For a comprehensive example of \$JOBUTIL capabilities, see the Program Preparation Example topic that follows.

## PROGRAM PREPARATION EXAMPLE

In the remainder of this section, a source module will be assembled, link edited, and formatted. Each step will first be treated separately, and then all steps will be combined under control of the batch job stream processor utility \$JOBUTIL.

## PROBLEM DESCRIPTION

In "Section 11. Terminal I/O", a program was developed, which, using a series of PRINTTEXT instructions, formatted a data entry screen (see the topic *Static Screen Coding Example* in Section 11). In "Section 14. Utility Programs," the \$IMAGE screen formatting utility was used to create the same screen, and to save it in a screen image data set named VIDEO1.

Supplied with the Event Driven Executive system are a group of supervisor subroutines which allow user programs to access stored screen images produced by \$IMAGE. The goal of this exercise is to replace the user-written formatting instructions (PRINTTEXTs) in the program developed in Section 11, with the appropriate subroutine calls to access the stored screen image in data set VIDEO1.

## Create/Modify Source Module

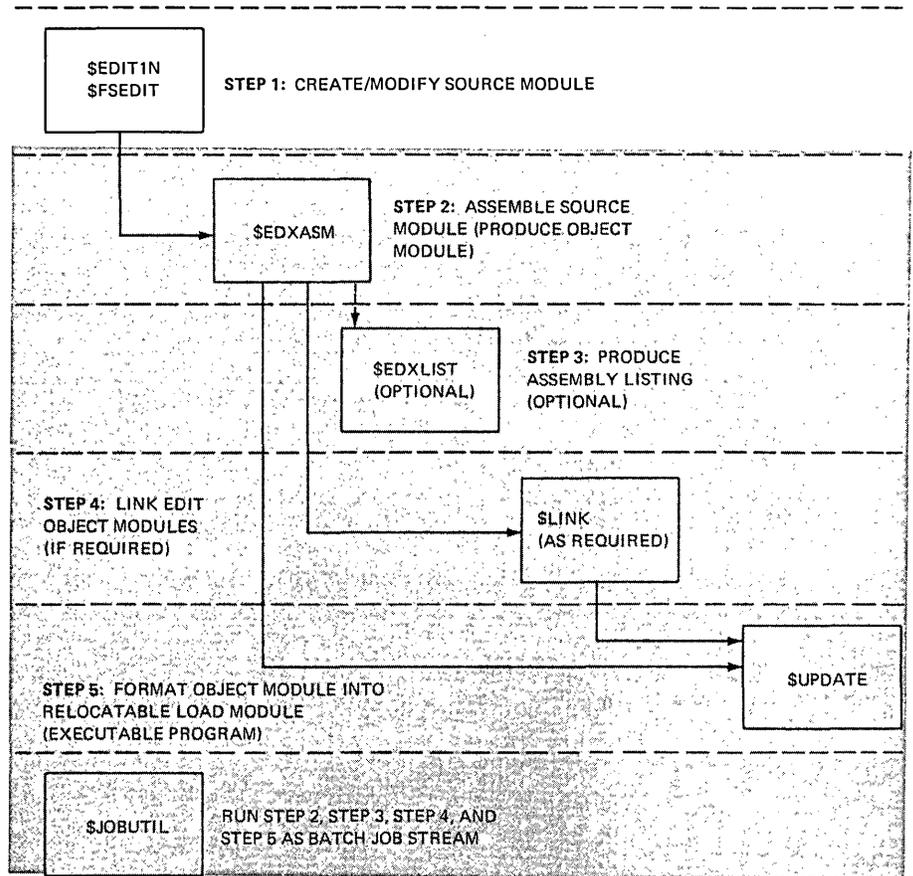


Figure 16-17. Step 1: Create source module

### Data Set Requirements.

#### UTILITY

<u>\$FSEDIT</u>	INPUT	OUTPUT	WORK	CONTROL
	DATA	DATA	DATA	DATA
<u>VOLUME</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>
EDX002		STATSRC	EDITWORK	
ASMVOL	SOURCE			

Figure 16-18. Data set requirements (1)

The source module to be modified is SOURCE on volume ASMVOL. Using \$FSEDIT, the program is read into the text edit work data set (Figure 16-19).

```

-----3--- SFSEEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

 1 BROWSE - DISPLAY DATASET
 2 EDIT   - CREATE OR CHANGE DATASET
 3 READ   - READ DATASET FROM HOST/NATIVE
 4 WRITE  - WRITE DATASET TO HOST/NATIVE
 5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
 6 LIST   - PRINT DATASET ON SYSTEM PRINTER
 7 MERGE  - MERGE DATA FROM A SOURCE DATASET
 8 END    - TERMINATE SFSEEDIT
 9 HELP   - DISPLAY TUTORIAL

-----

ENTER DATASET (NAME,VOLUME) : SOURCE,ASHVOL

```

Figure 16-19. Program preparation (1)

The screen formatting code begins at statement 140. In Figure 16-20, DD is entered to the left of statement 140, defining the start of a block delete.

```

EDIT --- EDITWORK, EDX002      75( 543)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 XMPLSTAT PROGRAM START
00020 IOCB1 IOCB NHIST=0
00030 IOCB2 IOCB SCREEN=STATIC
00040 ATTNLIST (END,OUT,$PF,STATIC)
00050 START ENQT IOCB1
00060 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00070 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00080 PRINTTEXT ' THE PROGRAM'
00090 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00100 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00110 DEQT
00120 CHECK WAIT ATTNECB,RESET
00130 IF (ATTNECB,EQ,1),GOTO,ENDIT
DD00140 ENTRY ENQT IOCB2
00150 ERASE MODE=SCREENTYPE=ALL
00160 TERMCTRL BLANK
00170 PRINTTEXT 'ENTER KEY = PAGE COMPLETE',LINE=1
00180 PRINTTEXT ' PF1 = DELETE ENTRY 1'
00190 PRINTTEXT ' PF2 = DELETE ENTRY 2'
00200 PRINTTEXT 'PF3 = DELETE ENTRY 3',SKIP=1
00210 PRINTTEXT 'PF4 = DELETE ENTRY 4'

```

Figure 16-20. Program preparation (2)

Scrolling down through the work area, the end of the formatting code is statement 370 where DD defines end of block delete.

```

EDIT --- EDITWORK, EDX002      75( 543)----- BLOCK COMMAND INCOMPLETE
COMMAND INPUT ==>                SCROLL ==> HALF
00220      PRINTX  DASHES,PROTECT=YES,LINE=3
00230      PRINTX  'CLASS NAME:',LINE=4,PROTECT=YES
00240      PRINTX  'INSTRUCTOR NAME:',LINE=4,PROTECT=YES,SPACES=32
00250 HDR    PRINTX  DASHES,PROTECT=YES,LINE=5
00260      MOVE    LINENBR,6
00270      DO      4,TIMES
00280      PRINTX  'NAME:',LINE=LINENBR,PROTECT=YES
00290      PRINTX  'STREET:',LINE=LINENBR,SPACES=30,PROTECT=YES
00300 A1     ADD    LINENBR,1
00310      PRINTX  'CITY:',LINE=LINENBR,SPACES=30,PROTECT=YES
00320 A2     ADD    LINENBR,1
00330      PRINTX  'STATE:',LINE=LINENBR,SPACES=30,PROTECT=YES
00340      ADD     LINENBR,3
00350      ENDDO
00360      PRINTX  LINE=4,SPACES=11
00370      TERMCTR DISPLAY
00380 WAITONE WAIT    KEY
00390      GOTO   (READ,E1,E2,E3,E4),X MPLSTAT+2
00400 E1     MOVE   LINENBR,6
00410      GOTO   DELETE
00420 E2     MOVE   LINENBR,11
00430      GOTO   DELETE

```

Figure 16-21. Program preparation (3)

After ENTER has been pressed and after you have scrolled back to the top of the data set, you will see the screen in Figure 16-22 with statements 140 through 370 deleted.

```

EDIT --- EDITWORK, EDX002      51( 543)----- COLUMNS 001 072
COMMAND INPUT ==>                SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 X MPLSTAT PROGRAM  START
00020 IOCB1  IOCB      NHIST=0
00030 IOCB2  IOCB      SCREEN=STATIC
00040      ATTNLIST  (END,OUT,SPF,STATIC)
00050 START  ENQT      IOCB1
00060      PRINTX  'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00070      PRINTX  'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00080      PRINTX  ' THE PROGRAM'
00090      PRINTX  'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00100      PRINTX  ' BRING UP THE ENTRY SCREEN'
00110      DEQT
00120 CHECK  WAIT      ATTNECB,RESET
00130      IF      (ATTNECB,EQ,1),GOTO,ENDIT
00380 WAITONE WAIT    KEY
00390      GOTO   (READ,E1,E2,E3,E4),X MPLSTAT+2
00400 E1     MOVE   LINENBR,6
00410      GOTO   DELETE
00420 E2     MOVE   LINENBR,11
00430      GOTO   DELETE
00440 E3     MOVE   LINENBR,16
00450      GOTO   DELETE

```

Figure 16-22. Program preparation (4)

By using the insert function of EDIT mode, the statements required to access the screen image in data set VIDEO1 can now be added.

## \$IMOPEN

### READING ASSIGNMENT: IBM Series/1 Event Driven Executive System Guide (SC34-0312), "Formatted Screen Images."

The first step in using a stored screen image is to read the image data set into the user program.

```
}
IMAGEBUF BUFFER      768,BYTES
DSETNAME TEXT       'VIDEO1,EDX002'
}
GETIMAGE CALL       $IMOPEN,(DSETNAME),(IMAGEBUF)
}
```

Figure 16-23. Program preparation (5)

Using subroutine \$IMOPEN, the data set is read into a user buffer. The name of the data set is specified in a TEXT statement, and the label of the TEXT statement is passed to \$IMOPEN as the first parameter in the CALL. The second parameter is the label of the buffer which will receive the image. Both parameters must be enclosed in parentheses.

The buffer is defined by a BUFFER statement, in bytes. Data set VIDEO1 is three records in length, so IMAGEBUF is defined as 768 bytes.

\$IMOPEN returns a completion code in "taskname+2", and it is a user responsibility to check for proper completion (-1 completion code). In Figure 16-24, the completion code check and error routine have been added.

```
.
.
IMAGEBUF BUFFER      768,BYTES
DSETNAME TEXT       'VIDEO1,EDX002'
.
.
.
GETIMAGE CALL       $IMOPEN,(DSETNAME),(IMAGEBUF)
IF                  (XMPLSTAT+2,NE,-1)
    MOVE            ERRCODE,XMPLSTAT+2
    PRINTTEXT      '@IMAGE OPEN ERROR, CODE ='
    PRINTNUM       ERRCODE
    GOTO           ERRQUERY
ENDIF
.
.
.
ERRCODE DATA       F'0'
ERRQUERY QUESTION   '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
.
.
```

Figure 16-24. Program preparation (6)

## *\$IMDEFN*

Before the screen can be displayed, the terminal must be enqueued as a static screen device. In Figure 16-25, the ENQT IOCB2 is preceded by a CALL to subroutine \$IMDEFN. This subroutine fills in the user-coded IOCB with the screen dimensions of the screen image in the buffer. The CALL to \$IMDEFN is not a required function; the IOCB may be enqueued without first calling the subroutine. By calling \$IMDEFN, you are assured that the IOCB will have the proper screen dimensions for the screen in the buffer. If \$IMAGE is used to change the dimensions of the stored screen image, the new dimensions will be placed in the IOCB by \$IMDEFN when the program next accesses that screen, with no change in the user program code required.

```

      .
      .
IMAGEBUF BUFFER      768,BYTES
DSETNAME TEXT      'VIDEO1,EDX002'
      .
      .
IOCB2      IOCB      SCREEN=STATIC
      .
      .
GETIMAGE CALL      $IMOPEN,(DSETNAME),(IMAGEBUF)
      IF          (XMPLSTAT+2,NE,-1)
      MOVE        ERRCODE,XMPLSTAT+2
      PRINTTEXT   '@IMAGE OPEN ERROR, CODE ='
      PRINTNUM    ERRCODE
      GOTO        ERRQUERY
      ENDIF
      CALL        $IMDEFN,(IOCB2),(IMAGEBUF)
      ENQT       IOCB2
      .
      .
ERRCODE DATA      F'0'
ERRQUERY QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT

```

Figure 16-25. Program preparation (7)

## *\$IMPROT/\$IMDATA*

Now that the terminal is enqueued, the screen image in the buffer can be displayed. In Figure 16-26, the TERMCTRL BLANK following the ENQT blanks the screen, preventing flicker while the image is written. The CALL of subroutine \$IMPROT transfers all the protected data from the image buffer to the screen, and the call to \$IMDATA transfers the unprotected data. (If a screen image consists of all protected or all unprotected data, only the appropriate subroutine need be called.)

```

      .
      .
IMAGEBUF BUFFER      768,BYTES
DSETNAME TEXT      'VIDE01,EDX002'
      .
      .
IOCB2      IOCB      SCREEN=STATIC
      .
      .
GETIMAGE CALL      $IMOPEN,(DSETNAME),(IMAGEBUF)
      IF          (XMPLSTAT+2,NE,-1)
      MOVE      ERRCODE,XMPLSTAT+2
      PRINTTEXT '@IMAGE OPEN ERROR, CODE ='
      PRINTNUM  ERRCODE
      GOTO      ERRQUERY
      ENDIF
      CALL      $IMDEFN,(IOCB2),(IMAGEBUF)
      ENQT      IOCB2
      TERMCTRL  BLANK
      CALL      $IMPROT,(IMAGEBUF),0
      CALL      $IMDATA,(IMAGEBUF)
      PRINTTEXT LINE=4,SPACES=11
      TERMCTRL  DISPLAY
      .
      .
ERRCODE DATA      F'0'
ERRQUERY QUESTION  '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT

```

Figure 16-26. Program preparation (8)

The PRINTTEXT following the last CALL positions the cursor at the first data entry field, and TERMCTRL DISPLAY unblanks the screen.

The second parameter of the CALL \$IMPROT statement (Figure 16-26) is coded as 0. This could be coded as the label of a BUFFER statement, in which case the \$IMPROT subroutine will build a table of the location and sizes of all unprotected (data entry) fields on the screen. Each table entry is three words in length. The first word will contain the line number and the second, the starting position of the field within the line (spaces from left margin of screen). The third word will contain the length of the field. These entries can be used to read/write data entry fields on the screen.

The "\$IM" subroutines are supplied as object modules. Because they are object modules, they are combined with the user program in the link edit step, not during assembly. They must therefore be declared as external references in an EXTRN statement.

Figures 16-27 and 16-28 are listings of the edit work data set after the edit session is complete. The EXTRN statement is statement 20, with the image buffer and screen image data set name definition following at 30 and 40. Other added statements include the "\$IM" code from 170 to 300, and the two statements at 670 and 680. The source module modification is complete. The work data set is written to STATSRC on volume EDX002 (\$FSEDIT Primary Option 4), completing Step 1 of the program preparation process.

```

00010 XMPLSTAT PROGRAM START
00020 EXTRN $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
00030 IMAGEBUF BUFFER 768,BYTES
00040 DSETNAME TEXT 'VIDEO1,EDX002'
00050 IOCB1 IOCB NHIST=0
00060 IOCB2 IOCB SCREEN=STATIC
00070 ATTNLIST (END,OUT,$PF,STATIC)
00080 START ENQT IOCB1
00090 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00100 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00110 PRINTTEXT ' THE PROGRAM'
00120 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00130 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00140 DEQT
00150 CHECK WAIT ATTNECB,RESET
00160 IF (ATTNECB,EQ,1),GOTO,ENDIT
00170 GETIMAGE CALL $IMOPEN,(DSETNAME),(IMAGEBUF)
00180 IF (XMPLSTAT+2,NE,-1)
00190 MOVE ERRCODE,XMPLSTAT+2
00200 PRINTTEXT '@IMAGE OPEN ERROR,CODE ='
00210 PRINTNUM ERRCODE
00220 GOTO ERRQUERY
00230 ENDIF
00240 CALL $IMDEFN,(IOCB2),(IMAGEBUF)
00250 ENQT IOCB2
00260 TERMCTRL BLANK
00270 CALL $IMPROT,(IMAGEBUF),0
00280 CALL $IMDATA,(IMAGEBUF)
00290 PRINTTEXT LINE=4,SPACES=11
00300 TERMCTRL DISPLAY
00310 WAITONE WAIT KEY
00320 GOTO (READ,E1,E2,E3,E4),XMPLSTAT+2
00330 E1 MOVE LINENBR,6
00340 GOTO DELETE
00350 E2 MOVE LINENBR,11
00360 GOTO DELETE
00370 E3 MOVE LINENBR,16
00380 GOTO DELETE
00390 E4 MOVE LINENBR,21
00400 DELETE ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
00410 ADD LINENBR,1
00420 ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
00430 ADD LINENBR,1
00440 ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR
00450 SUBTRACT LINENBR,2
00460 PRINTTEXT LINE=LINENBR,SPACES=5
00470 TERMCTRL DISPLAY
00480 GOTO WAITONE
00490 READ QUESTION 'MORE ENTRIES ?',LINE=2,SPACES=55,NO=CLEANUP

```

Figure 16-27. Program preparation (10)

```

00500          ERASE          MODE=LINE,LINE=2,SPACES=55,TYPE=DATA
00510          ERASE          MODE=SCREEN,LINE=6
00520          PRINTTEXT     LINE=6,SPACES=5
00530          TERMCTRL     DISPLAY
00540          GOTO          WAITONE
00550 CLEANUP  ERASE          MODE=SCREEN,TYPE=ALL
00560          DEQT
00570          GOTO START
00580 ENDIT    PROGSTOP
00590          DATA          X'5050'
00600 DASHES   DATA          80C'- '
00610 OUT     POST           ATTNECB,1
00620          ENDATTN
00630 STATIC  POST           ATTNECB,-1
00640          ENDATTN
00650 ATTNECB ECB
00660 LINENBR DATA          F'0'
00670 ERRCODE DATA          F'0'
00680 ERRQUERY QUESTION     '@RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT
00690          ENDPROG
00700          END

```

Figure 16-28. Program preparation (10 continued)

```

-----4-- SFSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

1 BROWSE - DISPLAY DATASET
2 EDIT - CREATE OR CHANGE DATASET
3 READ - READ DATASET FROM HOST/NATIVE
4 WRITE - WRITE DATASET TO HOST/NATIVE
5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
6 LIST - PRINT DATASET ON SYSTEM PRINTER
7 MERGE - MERGE DATA FROM A SOURCE DATASET
8 END - TERMINATE SFSEDIT
9 HELP - DISPLAY TUTORIAL

-----

ENTER DATASET (NAME,VOLUME) : 

```

Figure 16-29. Program preparation (11)

# Assemble Source Module

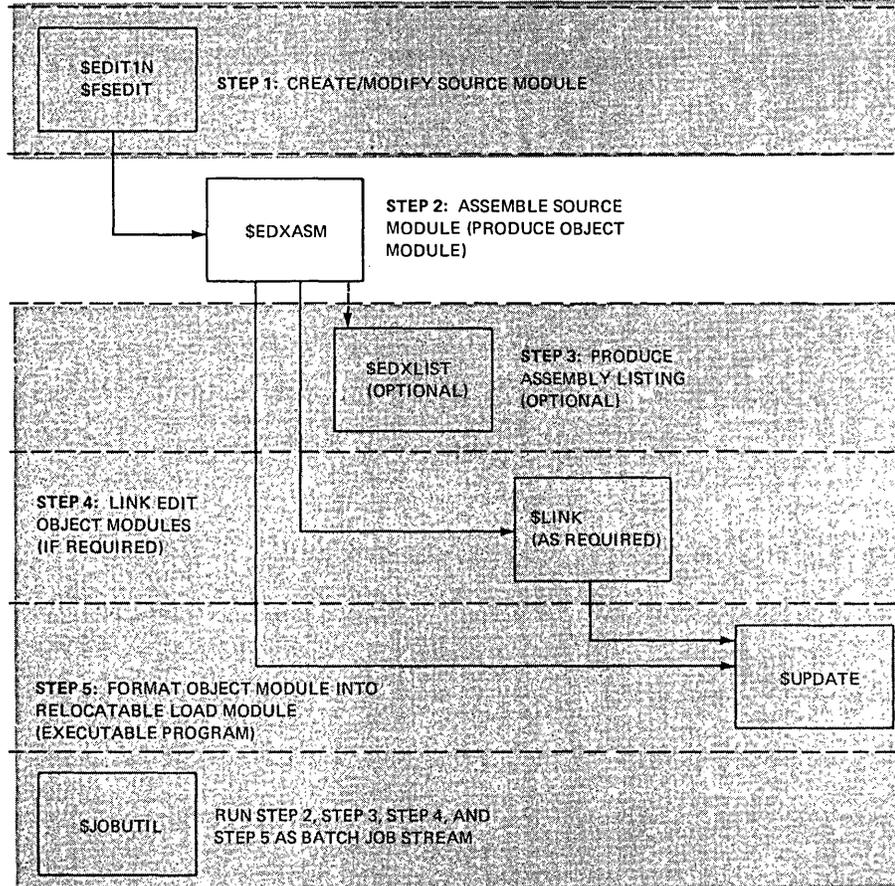


Figure 16-30. Step 2: Assemble source module

## Data Set Requirements

**UTILITY**

	INPUT DATA SET	OUTPUT DATA SET	WORK DATA SET	CONTROL DATA SET
<u>SEDXASM</u>				
<u>VOLUME</u>				
EDX002	STATSRC	ASMOUT	ASMWORK	
ASMLIB				\$EDXL
ASMVOL				

Figure 16-31. Data set requirements (12)

In Figure 16-32, the load request for the assembler is entered. Since the prompting sequence for the data set required by the assembler is known, these data set names are entered as advance input on the same line as the input request.

```
> $L $EDXASM,ASMLIB STATSRC ASMWORK ASMOUT  
$EDXASM 76P,03:14:35, LP= 7F00  
  
SELECT OPTIONS (?): END
```

Figure 16-32. Program preparation (12)

Because no options are selected, a full listing will be produced on the system printer, and the language control data set used for this assembly will be \$EDXL. When the assembler finishes, the resulting object module will be stored in ASMOUT on volume EDX002. \$EDXASM will then load \$EDXLIST to produce the assembly listing.

## Produce Assembly Listing

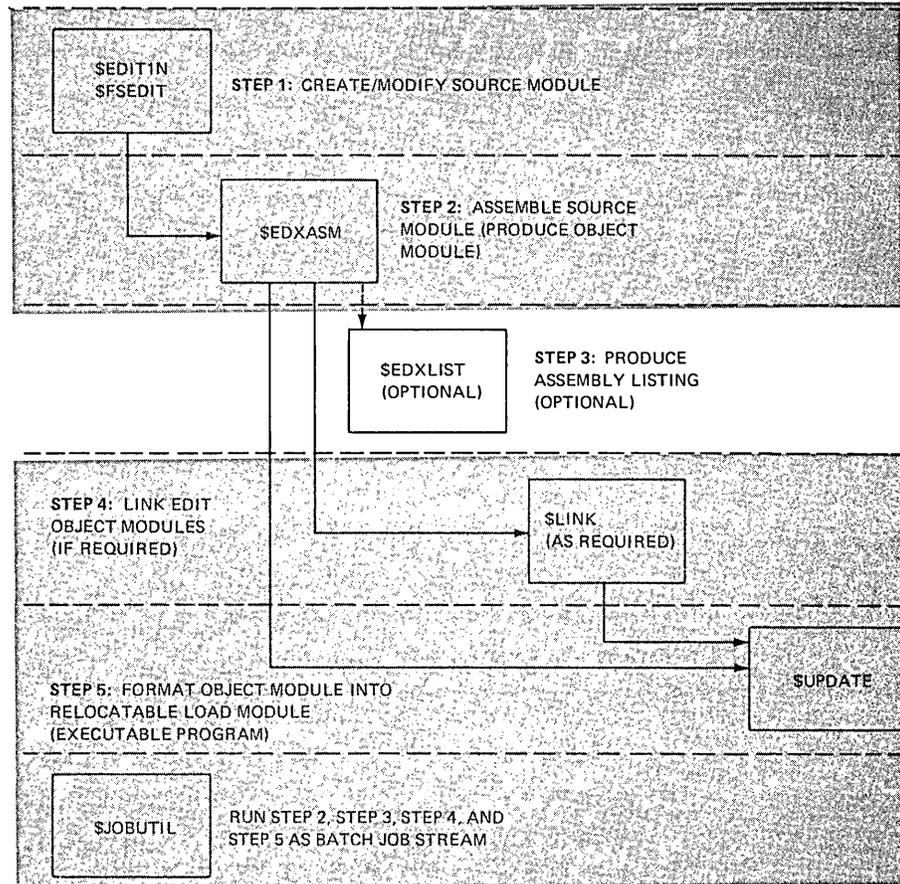


Figure 16-33. Step 3: Produce assembly listing

## Data Set Requirements

UTILITY				
<u>\$EDXLIST</u>	INPUT	OUTPUT	WORK	CONTROL
<u>VOLUME</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>
EDX002	ASMWORK	STATSRC		
ASMLIB				\$EDXL

Figure 16-34. Data set requirements (3)

In this example, \$EDXLIST is loaded by \$EDXASM. If the response to the SELECT OPTIONS (?): prompt had been NOLIST, \$EDXLIST would not have been invoked by \$EDXASM, but can still be loaded as a separate program by the operator. For example, if NOLIST were selected, and the assembly statistics displayed on the loading terminal at the end of the assembly indicated that there were assembly errors, \$EDXLIST can then be loaded to print a listing. \$EDXLIST will prompt for the source data set and the assembler work data set, and will get the name of the language control data set from the work data set, in which it is stored, at the end of the assembly. As long as an intervening assembly has not altered the contents of the assembler work data set, and you have not modified the source or language control data sets, \$EDXLIST will produce the same listing when loaded by the terminal operator after an assembly as it would were it loaded by \$EDXASM as part of the assembly step.

The assembly listing produced by the assembly requested in Figure 16-32 is shown in Appendix B, Figure B-1.

### Link Edit Object Modules

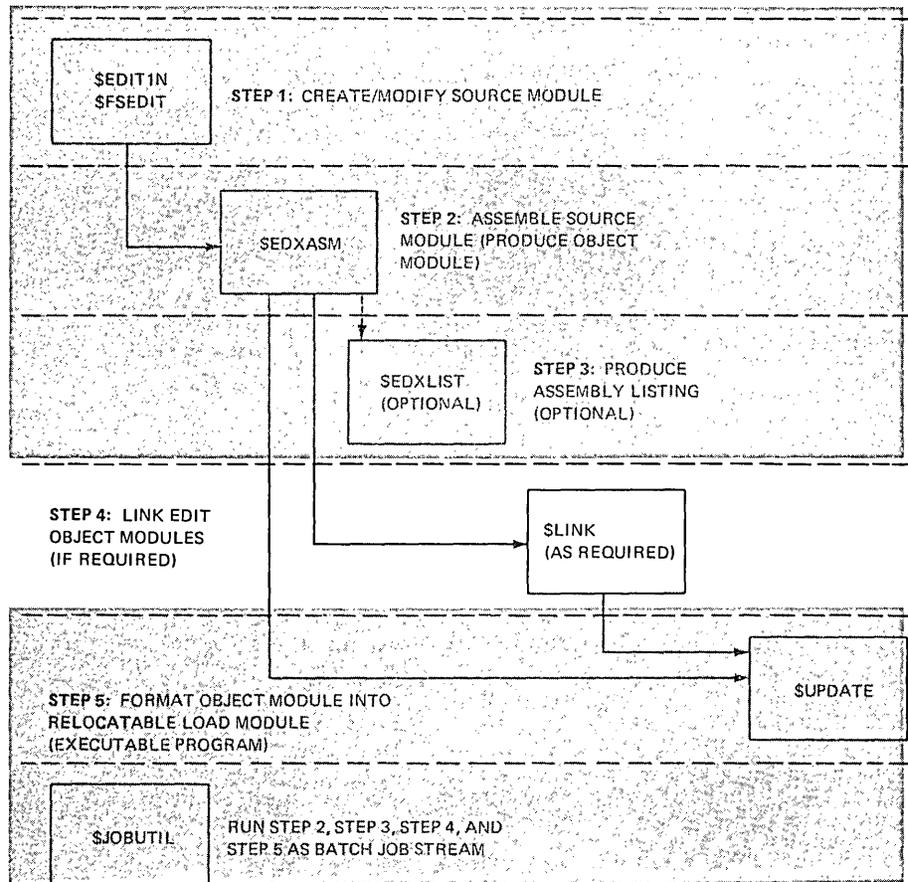


Figure 16-35. Step 4: Link edit object modules

Data Set Requirements

**UTILITY**

<u>\$LINK</u>	INPUT DATA <u>SET</u>	OUTPUT DATA <u>SET</u>	WORK DATA <u>SET</u>	CONTROL DATA <u>SET</u>
EDX002	ASMOUT	LINKOUT	LINKWRK1 LINKWRK2	LINKSTAT
ASMLIB	\$AUTO \$LEMSG \$IMGEN \$IMOPEN			

Figure 16-36. Data set requirements (4)

The screen formatting subroutines (\$IMOPEN, \$IMDEFN, \$IMDATA, \$IMPROT) used by the source program are distributed in the form of object modules. To include these subroutines in the program, the object module output of the assembly (data set ASMOUT) must be linked with the screen formatting support object modules.

Instead of requiring that INCLUDE control records for the screen formatting object modules be user-defined, they are system-defined in the system-supplied autocall data set \$AUTO, and may be included using the autocall option.

```

00010 $GPLIST,ASMLIB      $GPLIST
00020 $PUHC,ASMLIB      $PUHC
00030 $GEPM,ASMLIB      $GEPM
00040 $GEAC,ASMLIB      $GEAC
00050 $$GIN,ASMLIB      $$GIN
00060 $PUFC,ASMLIB      $PUFC
00070 $PUXC,ASMLIB      $PUXC
00080 $GEER,ASMLIB      $GEER
00090 $GEXC,ASMLIB      $GEXC
00100 $$SCREEN,ASMLIB   $$SCREEN
00110 $PUIC,ASMLIB      $PUIC
00120 $PUSC,ASMLIB      $PUSC
00130 $GESC,ASMLIB      $GESC
00140 $GEFC,ASMLIB      $GEFC
00150 $PUAC,ASMLIB      $PUAC
00160 $PUEC,ASMLIB      $PUEC
00170 $GEIC,ASMLIB      $GEIC
00180 $$PGIN,ASMLIB     $$PGIN
00190 $$CONCAT,ASMLIB   $$CONCAT
00200 $$XYPLT,ASMLIB    $$XYPLT
00210 $MFSL,ASMLIB      $MFSL
00220 $IMDEFN,ASMLIB    $IMDEFN      $IMPROT      $IMDATA
00230 $PACK,ASMLIB      $PACK
00240 $UNPACK,ASMLIB    $UNPACK
00250 $IMOPEN,ASMLIB    $IMOPEN      $OSUPEN
00260 $$RETURN,ASMLIB   RETURN
00270 $$SVC,ASMLIB      SVC
00280 $IMDTYP,ASMLIB    $IMDTYP
00290 $EXATSR,ASMLIB    SETBUSY      SUPEXIT      **END

```

Figure 16-37. Program preparation (13)

Figure 16-37 is a listing of \$AUTO, the system-supplied autocall data set. The screen formatting support modules are specified in autocall definition statements 220 and 250.

If you wished to have your own autocall definitions, you could add them to this data set, and continue to use the system-supplied autocall data set \$AUTO, or build your own autocall data set. In either case, the last statement in the data set must contain the "\*\*\*END" text, indicating the end of the autocall data set.

The output object module data set, the autocall data set (if required), and the object modules to be linked are passed to the link editor in the link control data set. The link control data set used for this example is named LINKSTAT. In Figure 16-38, the link control statements required for this link edit are listed, along with some preceding comment lines explaining their function.

```
00010 * THIS LINK EDIT CONTROL DATA SET SPECIFIES:
00020 *      1) THE LINKED OUTPUT OBJECT MODULE WILL
00030 *          BE STORED IN 'LINKOUT' ON EDX002
00040 *      2) THE AUTOCALL DATA SET IS '$AUTO' ON
00050 *          VOLUME ASMLIB (SYSTEM SUPPLIED)
00060 *      3) 'ASMOUT' ON EDX002 IS THE ONLY INPUT
00070 *          OBJECT MODULE TO BE INCLUDED
00080 *
00090 OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
00100 INCLUDE ASMOUT
00110 END
```

Figure 16-38. Program preparation (14)

This control statement file is created using \$EDIT1N or \$FSEEDIT, and stored in LINKSTAT using the SAVE/WRITE function at the end of the text edit session.

```
> $L $LINK,EDX002 LINKSTAT LINKWRK1 LINKWRK2
$LINK          76P,03:31:45, LP= 7F00

ENTER DEVICE NAME FOR PRINTED OUTPUT
$SYSPRTR
```

Figure 16-39. Program preparation (15)

At \$LINK load time, the operator supplies the name of the link control data set and the two link edit work data sets, along with the name of the device to which link edit messages will be directed. The link editor, using the LINKSTAT link control data set, links the assembled object module in ASMOUT (INCLUDE control statement) with screen formatting object modules in ASMLIB, found through autocall definitions in \$AUTO; the linked object module is stored in LINKOUT (OUTPUT control statement). Required error or information messages are read from the system-supplied link message data set, \$LEMSG.

See Appendix B, Figure B-2 for the \$SYSPRTR output resulting from this link edit.

## Format Object Module

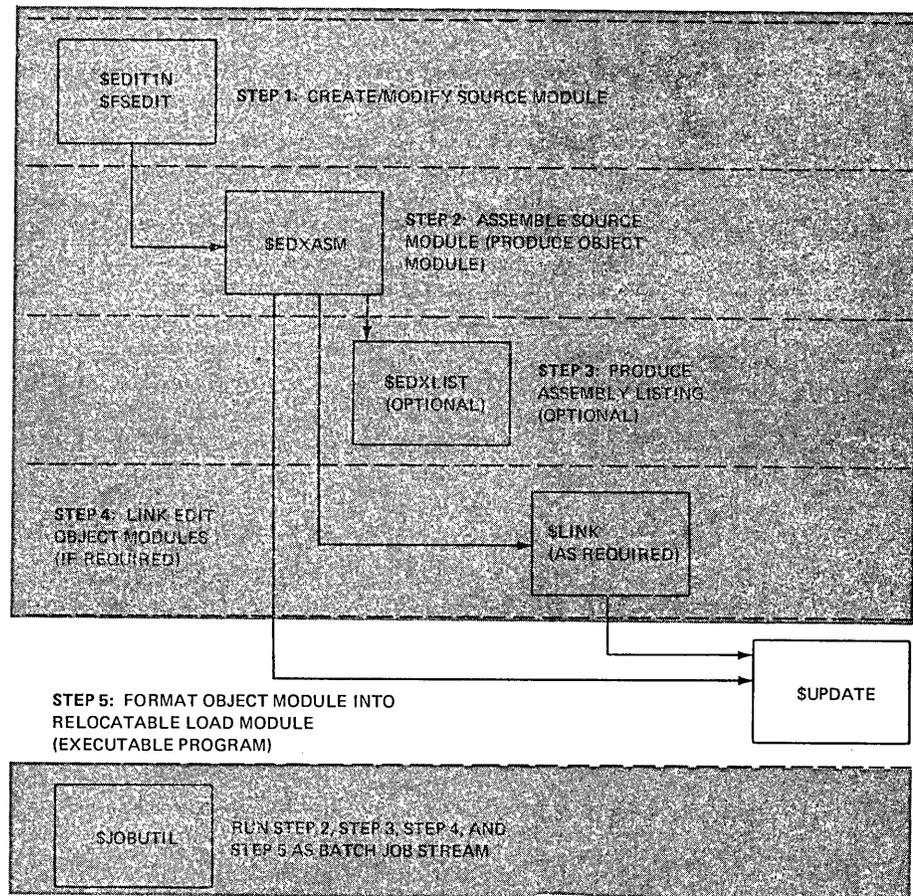


Figure 16-40. Step 5: Format object module

*Data Set Requirements*

**UTILITY**

<u>\$UPDATE</u>	INPUT	OUTPUT	WORK	CONTROL
	DATA	DATA	DATA	DATA
<u>VOLUME</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>	<u>SET</u>
EDX002	LINKOUT	STATPROG		

Figure 16-41. Data set requirements (5)

Before a linked (or assembled) object module can be executed, it must first be processed by \$UPDATE. This utility formats the object module into a relocatable load module, acceptable to the system loader.

```
> $L $UPDATE
$update 33P,03:33:10, LP= 7F00

THE DEFINED INPUT VOLUME IS EDX002, OK? Y
THE DEFINED OUTPUT VOLUME IS EDX002, OK? Y

COMMAND (?): RP LINKOUT STATPROG
```

Figure 16-42. Program preparation (16)

The "RP" command means "Read Program", and is followed by the name of the object module to be formatted, and the name of the resulting executable program. If data set STATPROG is not already allocated, \$UPDATE will create it. The program STATPROG can be loaded and executed when this step is completed.

## **\$EDXASM Copy Code Function**

In the discussion of the link edit step, object modules were automatically included in the link edit, using the autocall feature of \$LINK. In a somewhat similar manner, source statements may be merged into a source module at assembly time, using the "copycode" capability of \$EDXASM.

During the assembly operation, \$EDXASM uses a language control data set. Figure B-3 in Appendix B is a listing of the system-supplied language control data set \$EDXL. This data set consists of three main parts. Statements 10 through 2520 are error messages that may be required during assembly. Statements 2530 through 2880 are \*OVERLAY definitions. These are special control statements, used by the system loader to find the appropriate assembler overlay for each source instruction encountered during an assembly.

The third section consists of the two \*COPYCOD definitions, statements 2890 and 2900. \$COPYCOD statements define logical volumes which may contain source data sets used as "copycode" source modules. The logical end of the language control data set is the \*\*STOP\*\*, statement 2910.

The system-supplied language control data set, \$EDXL, has volumes ASMLIB and EDX002 defined as copycode volumes. When a COPY statement specifying the name of a source data set is encountered during the assembly of a source module, \$EDXASM will search ASMLIB and EDX002 for a data set of that name, and will include the source statements in that data set in the assembly, if found. User source data sets stored on ASMLIB or EDX002 may be used as copycode modules in assemblies using \$EDXL for a language control data set. If copycode data sets reside on other logical volumes, \$EDXL must be modified (\*COPYCOD statements added) to define those volumes to \$EDXASM as copycode volumes, or a user-defined language control data set containing the new \*COPYCOD definitions must be used for the assembly. A user-defined language control data set might be preferred to avoid altering \$EDXL.

Figures 16-43 through 16-50 will illustrate how to set up a user-defined language control data set, and how to code the COPY function in a user program.

In Figures 16-43 through 16-45, the system-supplied language control data set, \$EDXL, is modified to establish volume EDX003 as a copycode volume. The modified version is stored in the user-defined language control data set STATEDXL, leaving \$EDXL undisturbed. Using \$FSEEDIT, the system-supplied language control data set \$EDXL is read into the edit work data set, and EDIT mode (Primary Option 2) is entered. After scrolling to the bottom of the data set, the screen in Figure 16-43 is displayed.

```

EDIT --- $EDXL , ASMLIB 291( 1089)----- COLUMNS 001 072
COMMAND INPUT ==> SCROLL ==> HALF
02720 *OVERLAY $ASMO00B ASMLIB SBIO IODEF
02730 *OVERLAY $ASMO00C ASMLIB FIND FINDNOT
02740 *OVERLAY $ASMO00D ASMLIB FPCONV FADD FSUB FMULT FDIVD
02750 *OVERLAY $ASMO00E ASMLIB PRINTNUM GETVALUE READTEXT PRINTTEXT CONVTB
02760 CONVTD
02770 *OVERLAY $ASMO00G ASMLIB PLOTGIN GIN SCREEN XYPLOT YTPLOT
02780 CONCAT TP STATUS
02790 *OVERLAY $ASMO00H ASMLIB BSCREAD BSCWRITE BSCOPEN BSCCLOSE BSCIOCB
02800 BSCLINE
02810 *OVERLAY $ASMO00I ASMLIB FORMAT
02820 *OVERLAY $ASMO00Q ASMLIB FIRSTQ LASTQ NEXTQ DEFINEQ
02830 *OVERLAY $ASMEXIO ASMLIB EXIODEV IDCBC DCB EXOPEN EXIO
02840 *OVERLAY $ASMO00S ASMLIB SYSTEM STOREMAP DISK TIMER TAPE
02850 *OVERLAY $ASMO00T ASMLIB TERMINAL
02860 *OVERLAY $ASMO00U ASMLIB HOSTCOMM SENSORIO DBBSIO GETMAIN FREEMAIN
02870 *OVERLAY $ASMO00F ASMLIB ASMERROR $IDEF OTE SLE
02880 *OVERLAY $ASMO00M ASMLIB WHEREAS TCBGET TCBPUT
02890 *COPYCOD ASMLIB
02900 *COPYCOD EDX002
02910 **STOP**
***** **** BOTTOM OF DATA *****

```

Figure 16-43. Program preparation (17)

Using the insert line command, a copycode definition is placed in front of the **\*\*STOP\*\*** statement.

```

EDIT --- $EDXL , ASMLIB 291( 1089)----- COLUMNS 001 072
COMMAND INPUT ==> SCROLL ==> HALF
02720 *OVERLAY $ASMO00B ASMLIB SBIO IODEF
02730 *OVERLAY $ASMO00C ASMLIB FIND FINDNOT
02740 *OVERLAY $ASMO00D ASMLIB FPCONV FADD FSUB FMULT FDIVD
02750 *OVERLAY $ASMO00E ASMLIB PRINTNUM GETVALUE READTEXT PRINTTEXT CONVTB
02760 CONVTD
02770 *OVERLAY $ASMO00G ASMLIB PLOTGIN GIN SCREEN XYPLOT YTPLOT
02780 CONCAT TP STATUS
02790 *OVERLAY $ASMO00H ASMLIB BSCREAD BSCWRITE BSCOPEN BSCCLOSE BSCIOCB
02800 BSCLINE
02810 *OVERLAY $ASMO00I ASMLIB FORMAT
02820 *OVERLAY $ASMO00Q ASMLIB FIRSTQ LASTQ NEXTQ DEFINEQ
02830 *OVERLAY $ASMEXIO ASMLIB EXIODEV IDCBC DCB EXOPEN EXIO
02840 *OVERLAY $ASMO00S ASMLIB SYSTEM STOREMAP DISK TIMER TAPE
02850 *OVERLAY $ASMO00T ASMLIB TERMINAL
02860 *OVERLAY $ASMO00U ASMLIB HOSTCOMM SENSORIO DBBSIO GETMAIN FREEMAIN
02870 *OVERLAY $ASMO00F ASMLIB ASMERROR $IDEF OTE SLE
02880 *OVERLAY $ASMO00M ASMLIB WHEREAS TCBGET TCBPUT
02890 *COPYCOD ASMLIB
02900 *COPYCOD EDX002
..... *COPYCOD EDX003
02910 **STOP**
***** **** BOTTOM OF DATA *****

```

Figure 16-44. Program preparation (18)

EDX003 is now defined as a copycode volume. The edit work data set is now written into data set STATEDXL, which was previously allocated for this purpose.

```
-----4-- $FSEDIT PRIMARY OPTION MENU -----
SELECT OPTION ==>

 1 BROWSE - DISPLAY DATASET
 2 EDIT   - CREATE OR CHANGE DATASET
 3 READ   - READ DATASET FROM HOST/NATIVE
 4 WRITE  - WRITE DATASET TO HOST/NATIVE
 5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
 6 LIST   - PRINT DATASET ON SYSTEM PRINTER
 7 MERGE  - MERGE DATA FROM A SOURCE DATASET
 8 END    - TERMINATE $FSEDIT
 9 HELP   - DISPLAY TUTORIAL

-----

ENTER DATASET (NAME,VOLUME) : STATEDXL.EDX002
```

Figure 16-45. Program preparation (19)

In Figures 16-46 and 16-47, a portion of code is extracted from the source data set STATSRC and stored on volume EDX003 in a data set named ROLL. This data set will be used as a copycode module.

Again using \$FSEDIT, the roll screen instructions from STATSRC are read into the work area, and identifying comments inserted at the beginning and end of the data set. This is accomplished by:

1. READ (Primary Option 3) STATSRC into work data set,
2. EDIT (Primary Option 2) and block delete statements 10 through 70, then statements 150 through 700 leaving only the "roll screen" statements
3. Insert comments at top and bottom, resulting in the screen shown in Figure 16-46.

```
EDIT --- EDITWORK, EDX002 13( 243)----- COLUMNS 001,072
COMMAND INPUT ==> SCROLL ==>HALF
***** **** TOP OF DATA *****
00010 *
00020 * START OF "COPYCODE" MODULE
00030 *
00040 START ENQT IOCB1
00050 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00060 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00070 PRINTTEXT ' THE PROGRAM'
00080 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2
00090 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00100 DEQT
00110 *
00120 * END OF "COPYCODE" MODULE
00130 *
***** **** BOTTOM OF DATA *****
```

Figure 16-46. Program preparation (20)

Now the COPY CODE module is written to data set ROLL (Figure 16-47).

```

-----4--- SFSEDT PRIMARY OPTION MENU -----
SELECT OPTION ==>

 1 BROWSE - DISPLAY DATASET
 2 EDIT   - CREATE OR CHANGE DATASET
 3 READ   - READ DATASET FROM HOST/NATIVE
 4 WRITE  - WRITE DATASET TO HOST/NATIVE
 5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM
 6 LIST   - PRINT DATASET ON SYSTEM PRINTER
 7 MERGE  - MERGE DATA FROM A SOURCE DATASET
 8 END    - TERMINATE SFSEDT
 9 HELP   - DISPLAY TUTORIAL

-----

ENTER DATASET (NAME,VOLUME) : ROLL,EDX003

```

Figure 16-47. Program preparation (21)

In Figures 16-48 through 16-50, STATSRC is again read into the edit work area, and modified to use the COPY function.

In Figure 16-48, STATSRC has been read into the work data set, and EDIT mode has been entered.

```

EDIT --- EDITWORK, EDX002      70( 243)----- COLUMNS 001 072
COMMAND INPUT ==>              SCROLL ==> HALF
***** TOP OF DATA *****
00010 XMLPLSTAT PROGRAM START
00020 EXTRN SIMOPEN,$IMDEFN,$IMPROT,$IMDATA
00030 IMAGEBUF BUFFER 768,BYTES
00040 DSETNAME TEXT 'VIDEO1,EDX002'
00050 IOCB1 IOCB NHIST=0
00060 IOCB2 IOCB SCREEN=STATIC
00070 ATTNLIST (END,OUT,$PF,STATIC)
00080 START ENQT IOCB1
00090 PRINTTEXT 'CLASS ROSTER PROGRAM',SPACES=15,LINE=1
00100 PRINTTEXT 'HIT "ATTN" AND ENTER "END" TO END',SKIP=2
00110 PRINTTEXT ' THE PROGRAM'
00120 PRINTTEXT 'HIT ANY PROGRAM FUNCTION KEY TO ',SKIP=2
00130 PRINTTEXT ' BRING UP THE ENTRY SCREEN'
00140 DEQT
00150 CHECK WAIT ATTNECB,RESET
00160 IF (ATTNECB,EQ,1),GOTO,ENDIT
00170 GETIMAGE CALL SIMOPEN,(DSETNAME),(IMAGEBUF)
00180 IF (XMLPLSTAT+2,NE,-1)
00190 MOVE ERRCODE,XMPLSTAT+2
00200 PRINTTEXT '@IMAGE OPEN ERROR, CODE ='
00210 PRINTNUM ERRCODE

```

Figure 16-48. Program preparation (22)

The "DD" to the left of statement 80 and 140 will perform a block delete of the statements that will be brought in as copy code. In Figure 16-49, the ENTER key has been depressed, and the delete is done.

```

EDIT --- EDITWORK, EDX002      63( 243)----- COLUMNS 001 072
COMMAND INPUT ==>              SCROLL ==> HALF
***** ***** TOP OF DATA *****
00010 X MPLSTAT PROGRAM START
00020          EXTRN $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
00030 IMAGEBUF BUFFER 768,BYTES
00040 DSETNAME TEXT 'VIDEO1,ASMVOL '
00050 IOCB1 IOCB NHIST=0
00060 IOCB2 IOCB SCREEN=STATIC
00070          ATTNLIST (END,OUT,$PF,STATIC)
00150 CHECK WAIT ATTNECB,RESET
00160          IF (ATTNECB,EQ,1),GOTO,ENDIT
00170 GETIMAGE CALL $IMOPEN,(DSETNAME),(IMAGEBUF)
00180          IF (X MPLSTAT+2,NE,-1)
00190          MOVE ERRCODE,X MPLSTAT+2
00200          PRINTTEXT '@IMAGE OPEN ERROR, CODE ='
00210          PRINTNUM ERRCODE
00220          GOTO ERRQUERY
00230          ENDIF
00240          CALL $IMDEFN,(IOCB2),(IMAGEBUF)
00250          ENQT IOCB2
00260          TERMCTRL BLANK
00270          CALL $IMPROT,(IMAGEBUF),0
00280          CALL $IMDATA,(IMAGEBUF)

```

Figure 16-49. Program preparation (23)

In Figure 16-50, a COPY command is inserted, naming the copy code module ROLL. When the assembler encounters the COPY statement, it will go to the language control data set to find the copy code volume definitions and locate the data set containing the copy code module. The source statements in ROLL will be inserted at this point in the source module, and assembled as part of STATSRC.

```

***** ***** TOP OF DATA ***** HALF
00010 XMPLSTAT PROGRAM START
00020          EXTRN  $IMOPEN,$IMDEFN,$IMPROT,$IMDATA
00030 IMAGEBUF BUFFER 768,BYTES
00040 DSETNAME TEXT  'VIDEO1,EDX002'
00050 IOCB1  IOCB  NHIST=0
00060 IOCB2  IOCB  SCREEN=STATIC
00070          ATTNLIST (END,OUT,$PF,STATIC)
00071 *
00072          COPY  ROLL
00073 *
00150 CHECK  WAIT  ATTNECB,RESET
00160          IF    (ATTNECB,EQ,1),GOTO,ENDIT
00170 GETIMAGE CALL  $IMOPEN,(DSETNAME),(IMAGEBUF)
00180          IF    (XMPLSTAT+2,NE,-1)
00190          MOVE  ERRCODE,XMPLSTAT+2
00200          PRINTTEXT '@IMAGE OPEN ERROR, CODE ='
00210          PRINTNUM ERRCODE
00220          GOTO  ERRQUERY
00230          ENDIF
00240          CALL  $IMDEFN,(IOCB2),(IMAGEBUF)
00250          ENQT  IOCB2

```

Figure 16-50. Program preparation (24)

The edit work data set is saved back into STATSRC using the WRITE function (Primary Option 4), and the source module is ready for assembly.

## Job Stream Procedure

You have seen how, once a source module has been created (\$EDIT1N or \$FSEDIT), the assembler (\$EDXASM), linkage editor (\$LINK), and load module formatter (\$UPDATE) may each be invoked in turn, using the \$L facility. Using a procedure file and \$JOBUTIL, all three steps may be run as a single job stream.

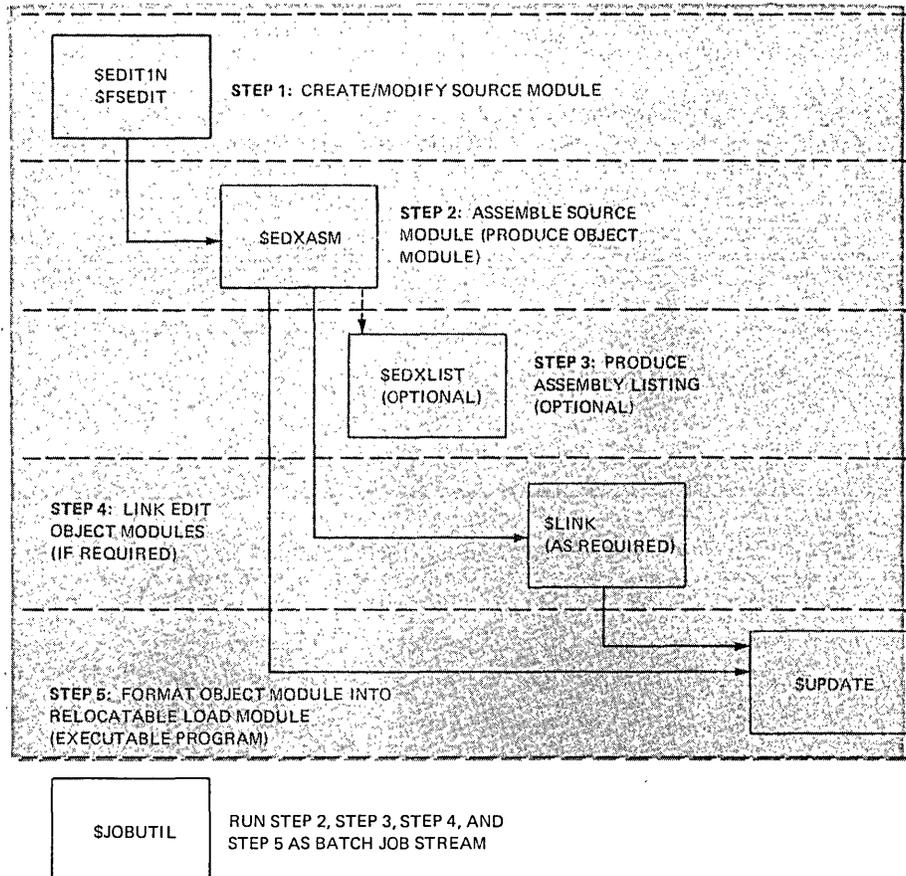


Figure 16-51. Job stream procedure

Appendix B, Figure B-4, is a listing of a batch job stream processor (\$JOBUTIL) procedure file. The statements in a procedure file are created using \$EDIT1N or \$FSEDIT, and saved in a data set. In this example, the procedure data set is STATPROC on EDX002.

When \$JOBUTIL is loaded, the operator is prompted for the name of a procedure file.

```

> $L $JOBUTIL
$JOBUTIL 4P,00:05:32, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): STATPROC
  
```

Figure 16-52. Program preparation (25)

In Appendix B, Figure B-4, the JOB command at statement 10 causes the display of a "job started" message on the loading terminal.

```
> $L $JOBUTIL
$JOBUTIL      4P,00:05:32, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): STATPROC
*** JOB - STATIC - STARTED AT 00:05:55 00/00/00 ***

JOB          STATIC
```

Figure 16-53. Program preparation (26)

The LOG command (statement 20, Figure B4) will cause the procedure file statements (other than internal comments) to print on the system printer. Statements 120 through 190 will load and execute the assembler. The source, work, and output data sets are specified in the DS commands. The PARM command at statement 170 directs the assembly listing to the system printer, and specifies STATEDXL as the language control data set for this assembly (STATEDXL contains the \*COPYCOD statement for volume EDX003, where ROLL is stored). The NOMSG command following the PARM prevents the \$EDXASM load message from being displayed on the loading terminal, but the REMARK at statement 130 will appear.

```
> $L $JOBUTIL
$JOBUTIL      4P,00:05:32, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): STATPROC
*** JOB - STATIC - STARTED AT 00:05:55 00/00/00 ***

JOB          STATIC
REMARK       ASSEMBLY OF 'STATSRC' STARTED
```

Figure 16-54. Program preparation (27)

The normal completion code for an error-free assembly is -1. The JUMP command (statement 200) tests the assembler completion code. If it is not equal to minus 1, the JUMP will transfer control to the label BADASM, which is defined by the LABEL command at statement 410. The REMARK at 420 would be displayed on the loading terminal, and the JUMP at 430 would transfer to label END, ending the job.

Assuming normal assembler operation, \$JOBUTIL would continue with statements through 350, the link edit step.

Through the PAUSE command, \$JOBUTIL allows input of job control commands by an operator. To illustrate this capability, the link control data set is not specified in a DS command. Instead, the PAUSE at statement 300 will allow entry of the link control data set name. When the link procedure is entered, the two REMARK statements preceding the PAUSE will be displayed, along with the PAUSE operator instructions, and \$JOBUTIL will wait for the operator to press ATTENTION and enter a command.

```
> $L $JOBUTIL
$JOBUTIL      4P,00:05:32, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): STATPROC
*** JOB - STATIC - STARTED AT 00:05:55 00/00/00 ***

JOB          STATIC
REMARK      ASSEMBLY OF 'STATSRC' STARTED
REMARK      LINK EDIT OF 'ASMOUT' OBJECT MODULE STARTED
REMARK      NAME OF LINK CONTROL DATA SET ?
```

PAUSE-\*--ATTN:GO/ENTER/ABORT

PAUSE

Figure 16-55. Program preparation (28)

The operator can continue (GO), enter a job control command (ENTER), or abort the job stream processor and end the job (ABORT). In this example, the operator wants to enter a command, so ENTER is requested. The operator is prompted for the command and the command operand. When GO is entered in response to the COMMAND prompt, \$JOBUTIL continues.

```

> $L $JOBUTIL
$JOBUTIL 4P,00:47:17, LP= 5F00
ENTER PROCEDURE (NAME,VOLUME): STATPROC
*** JOB - STATIC - STARTED AT 00:47:26 00/00/00 ***

JOB          STATIC
REMARK      ASSEMBLY OF 'STATSRC' STARTED
REMARK      LINK EDIT OF 'ASMOUT' OBJECT MODULE STARTED
REMARK      NAME OF LINK CONTROL DATA SET ?

PAUSE-*--ATTN:GO/ENTER/ABORT

PAUSE
> ENTER

ENTER COMMAND DS

ENTER OPERAND LINKSTAT

ENTER COMMAND GO

```

Figure 16-56. Program preparation (29)

**\$JOBUTIL** allows secondary or nested procedures to be invoked from a primary procedure. To illustrate, the formatting job control statements have been defined as a nested procedure, stored in data set FORMPROC.

```

00010 *****
00020 * THIS IS A "NESTED" PROCEDURE, INVOKED FROM
00030 * 'STATPROC' BY THE 'PROC' COMMAND. $JOBUTIL
00040 * SUPPORTS ONE LEVEL OF NESTING.
00050 *
00060 REMARK   FORMATTING OF 'LINKOUT' STARTED
00070 PROGRAM   $UPDATE
00080 PARM      $SYSPRTR LINKOUT  STATPROG YES
00090 NOMSG
00100 EXEC
00110 EOP

```

Figure 16-57. Program preparation (30)

The primary procedure (Appendix B, Figure B-4), after testing for a successful link edit (JUMP command at statement 360), invokes the nested procedure FORMPROC by the PROC command at statement 370. At the conclusion of the formatting step, control is returned to the primary procedure at statement 380. If \$UPDATE executed properly, the job is ended without displaying the error message (REMARK at 390).

```
> $L $JOBUTIL
ENTER PROCEDURE (NAME,VOLUME): STATPROC
*** JOB - STATIC - STARTED AT 00:05:55 00/00/00 ***

JOB      STATIC
REMARK   ASSEMBLY OF 'STATSRC' STARTED
REMARK   LINK EDIT OF 'ASMOUT' OBJECT MODULE STARTED
REMARK   NAME OF LINK CONTROL DATA SET ?

PAUSE-*--ATTN:GO/ENTER/ABORT

PAUSE
> ENTER

ENTER COMMAND DS

ENTER OPERAND LINKSTAT

ENTER COMMAND GO
REMARK   FORMATTING OF 'LINKOUT' STARTED

$JOBUTIL ENDED AT 00:10:18
```

Figure 16-58. Program preparation (31)

Figure B-5 in Appendix B is the \$SYSPRTR output resulting from execution of the \$JOBUTIL procedure file STATPROC, including the assembly listing with the ROLL copy code statements successfully merged.

**This page intentionally left blank.**

## Section 17. Program Preparation Using \$\$S1ASM

### OBJECTIVES

After completing this section, the student should be able to use \$\$S1ASM to assemble application programs written in Series/1 assembler and/or Event Driven language.

### \$\$S1ASM MACHINE READABLE MATERIAL

Licensed program 5719-ASA is distributed from PID on a diskette with a volume name of ASA001. Included on the diskette are the following components:

1. Series/1 macro assembler (\$\$S1ASM)
2. Linkage editor (\$LINK)
3. System definition file, procedure file, and link control file, for use in system generation using \$\$S1ASM (5719-LM7 Macro Library is a prerequisite for system generation using \$\$S1ASM).
4. Source, procedure, and link control files for an installation verification test program.

\$\$S1ASM, unlike \$EDXASM, is a macro assembler. It will assemble programs coded in Series/1 assembler language, such as USER exit routines, and when an operation code not in the Series/1 instruction set is encountered, it will search a macro library for a macro of that name. Licensed program 5719-LM7 is the macro library containing macro prototypes for all the Event Driven language statements, and must be installed if \$\$S1ASM is to be used for assembling Event Driven language programs, or to build tailored supervisors.

\$\$S1ASM runs under control of the Event Driven Executive supervisor, so the system on which \$\$S1ASM assemblies are run must also have the Event Driven Executive Basic Supervisor and Emulator installed. Program preparation aids and utilities are provided by the Event Driven Executive Utilities.

Output object modules produced by \$\$S1ASM assemblies must be processed by \$LINK before being formatted into executable load modules by \$UPDATE.

*Note:* For users who will be coding programs in the Event Driven language, the Program Preparation Facility is recommended. \$EDXASM is a direct assembler for the Event Driven language; no macro processing is involved, and therefore performance is much higher than with \$S1ASM and the 5719-LM7 Macro Library.

## INSTALLING \$S1ASM

Installation procedures for the Series/1 macro assembler are in the Program Directory, shipped with the program from PID. As with the other Event Driven Executive program offerings, the \$COPYUT1 utility is used to transfer the contents of PID diskettes to disk. If the installation instructions in the Program Directory are followed, \$S1ASM will reside on logical volume ASMLIB on disk.

## \$S1ASM OPERATION

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Utilities, Operator Commands, and Program Preparation (SC34-0313), "Program Preparation Using \$S1ASM."

Figure 17-1 outlines the steps required to prepare programs for execution using the \$S1ASM macro assembler.

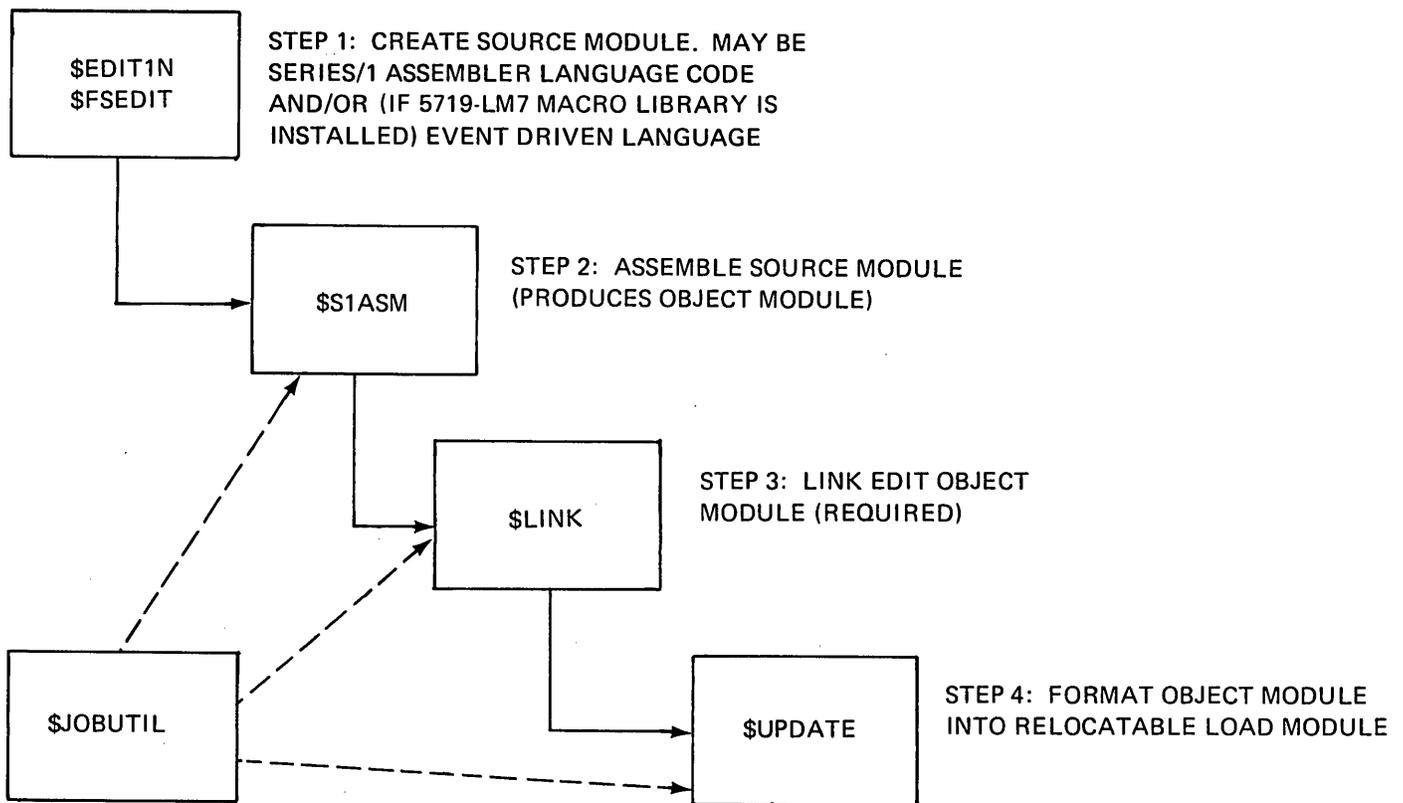


Figure 17-1. \$S1ASM program preparation overview

The program preparation steps illustrated in Figure 17-1 closely parallel those required for program preparation using \$EDXASM (see Section 16, Figure 16-1), but with the following differences:

1. The source module may contain Series/1 assembler language code and/or have references to user-written macros.
2. The link edit step is mandatory, even if the assembled program has no external references. The object module produced by a \$\$S1ASM assembly cannot be formatted into a load module by \$UPDATE without first being processed by \$LINK.

## Data Set Requirements

\$\$S1ASM uses three work data sets, which must be allocated by the user.

```
> $L $DISKUT1
$DISKUT1      37P,00:21:02, LP= 8300

USING VOLUME EDX002

COMMAND (?): AL ASAWORK1 2000
DEFAULT TYPE = DATA - OK? Y
ASAWORK1 CREATED

COMMAND (?): AL ASAWORK2 2000
DEFAULT TYPE = DATA - OK? Y
ASAWORK2 CREATED

COMMAND (?): AL ASAWORK3 800
DEFAULT TYPE = DATA - OK? Y
ASAWORK3 CREATED

COMMAND (?): END

$DISKUT1 ENDED AT 00:22:13
```

Figure 17-2. Allocate work files

The file sizes shown in Figure 17-2 are not unusually big for \$\$S1ASM work data sets, and may have to be increased to accommodate a large assembly. Note that WORK1 and WORK2 must be of equal size.

In addition to the three work files, \$\$S1ASM also requires:

- A source data set containing the statements to be assembled
- An object output data set in which the object module produced by the assembly will be stored
- If the source file contains macro references (Event Driven language statements or references to user-coded macros), at least one, and optionally two volume names must be supplied, on which reside the macro prototypes referenced in the source module

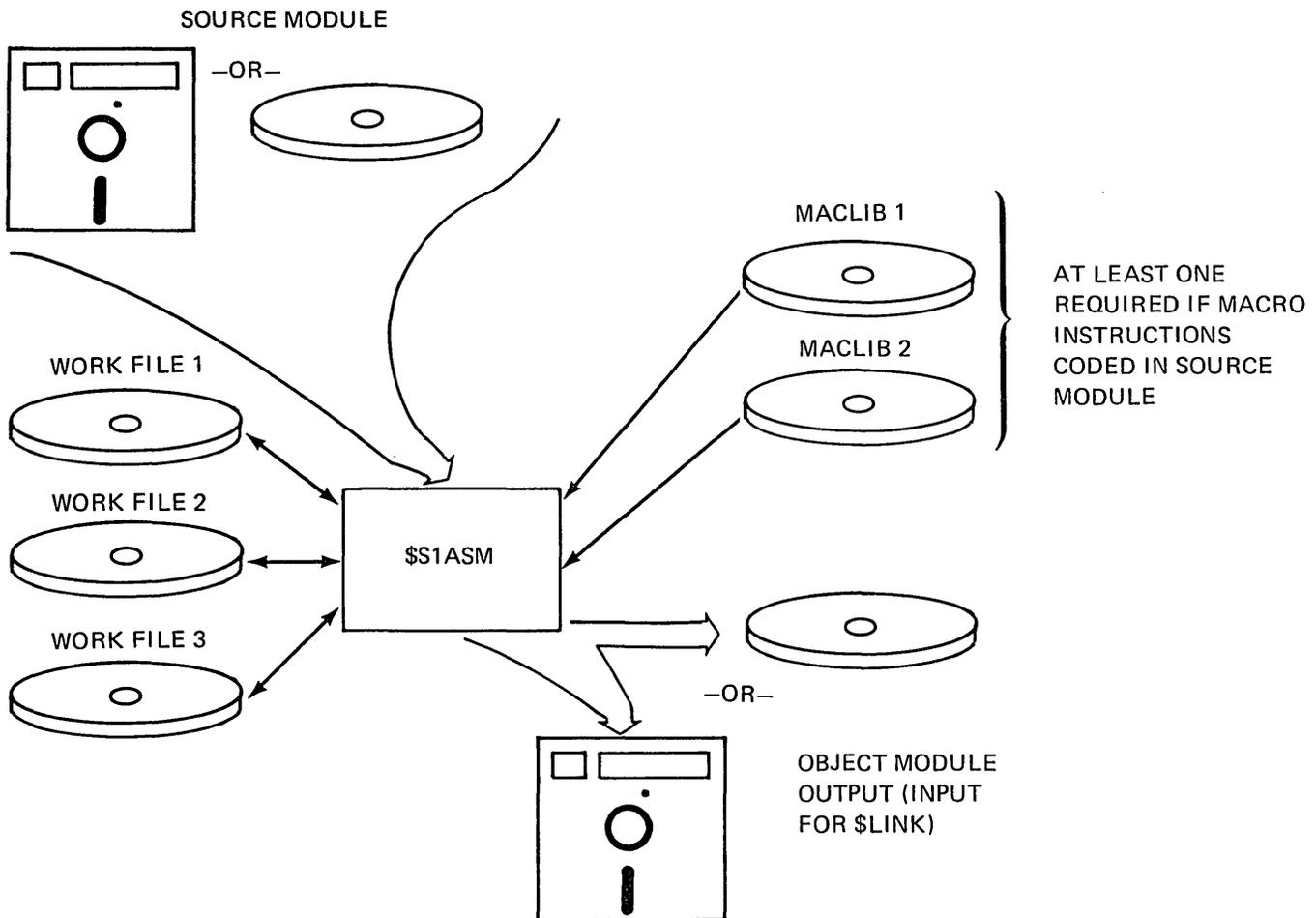


Figure 17-3. \$S1ASM data set requirements

At load time, the operator is prompted for source, work, and output data set names.

```

> $L $S1ASM,ASMLIB
SOURCE (NAME,VOLUME): STSOURCE
WORK1 (NAME,VOLUME): ASAWORK1
WORK2 (NAME,VOLUME): ASAWORK2
WORK3 (NAME,VOLUME): ASAWORK3
OBJECT (NAME,VOLUME): ASMOBJ
$S1ASM 88P,00:24:19, LP= 8300

MACLIB1 (?): EDX003
MACLIB2 (?):
ENTER OPTIONS (?):
ENTER OUTPUT DEVICE NAME:
CPA000I ASSEMBLER STARTED

```

Figure 17-4. \$S1ASM load

In Figure 17-4, volume EDX003 is specified as MACLIB1. When macro references are encountered in source data set S1SOURCE, volume EDX003 will be searched for the appropriate macro. Assuming S1SOURCE contains Event Driven language statements, EDX003 would contain the Event Driven Executive Macro Library.

If required, a second volume may be specified (MACLIB2 (?): prompt), in which case the second volume would be searched if a required macro were not found on the volume specified as MACLIB1.

In Figure 17-4, the default assembler options are taken, and the output is defaulted to \$SYSPRTR (null response to both prompts). With the default assembler options, \$S1ASM printed output can be voluminous, so you may wish to exercise options to suppress certain parts of the listing (see reading assignment for available options).

### \$S1ASM/\$JOBUTIL INTERFACE

\$S1ASM, like \$EDXASM, may be executed under control of the job stream processor, \$JOBUTIL, using job control statements in a procedure file. In Figure 17-5, the job control statements in the procedure file at the right are the equivalents of the operator response sequence at the left.

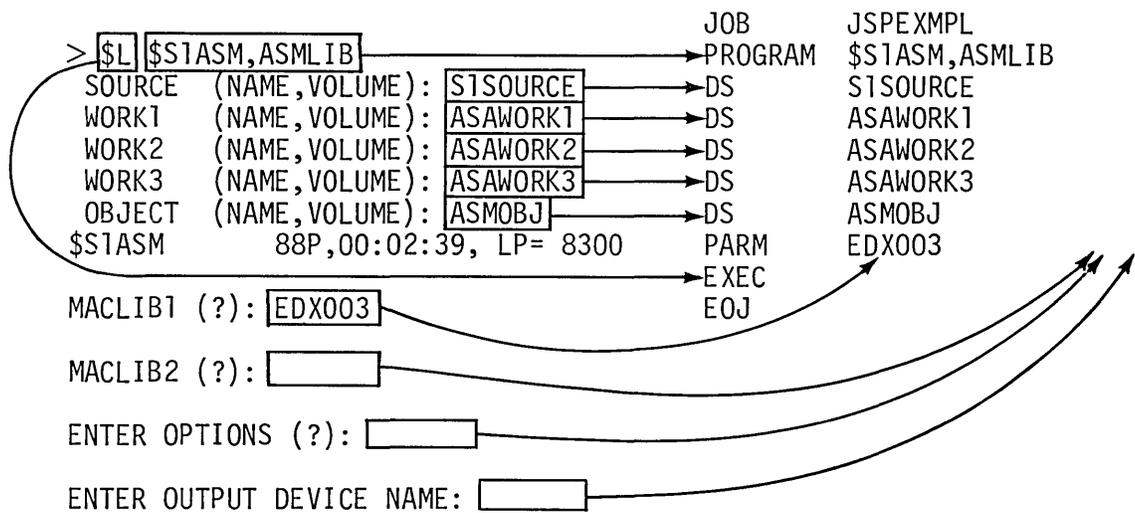


Figure 17-5. \$JOBUTIL procedure

All program preparation steps other than the actual assembly are identical for \$S1ASM and for \$EDXASM. See the appropriate topics in "Section 16. Program Preparation Using \$EDXASM" for information on creating a source module, the link edit step, and formatting a linked object module into a relocatable load module.



**This page intentionally left blank.**



## Section 18. Session Manager

### OBJECTIVES

At the conclusion of this section, the student should be able to:

1. Describe basic Session Manager operation
2. Use the Session Manager to run system utilities/program preparation programs

### SESSION MANAGER OVERVIEW

Throughout this study guide, you have seen numerous examples of the use of the \$L operator command to load programs and system utility programs, as illustrated in Figure 18-1.

```
> $L $UTILITY,volume dsname,volume
```

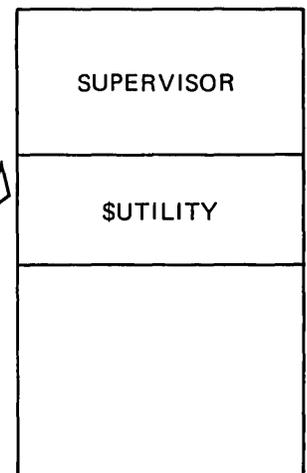


Figure 18-1. Load utility using \$L

In general, the following statements are true about the use of the \$L operator command:

1. All data sets required by the program or utility being loaded, using the \$L command, must be allocated by the user, prior to the load operation, using \$DISKUT1.
2. All data set names referenced by the program or utility to be loaded (which are not already specified in the DS= list of the PROGRAM statement) must be supplied by the operator each time the program or utility is loaded, even in a repetitive execution environment such as program assembly and debug.
3. All execution time options, such as output device, listing options, etc., are requested with each load of the program/utility, even if the responses are identical to previous executions and/or even if the default options (null entry) are acceptable.

In "Section 16. Program Preparation Using \$EDXASM", you were introduced to \$JOBUTIL, the job stream processor utility. By allocating (\$DISKUT1) and then creating (\$FSEEDIT, \$EDIT1N) a job control procedure data set, a job can be run under control of \$JOBUTIL, as illustrated in Figure 18-2 below.

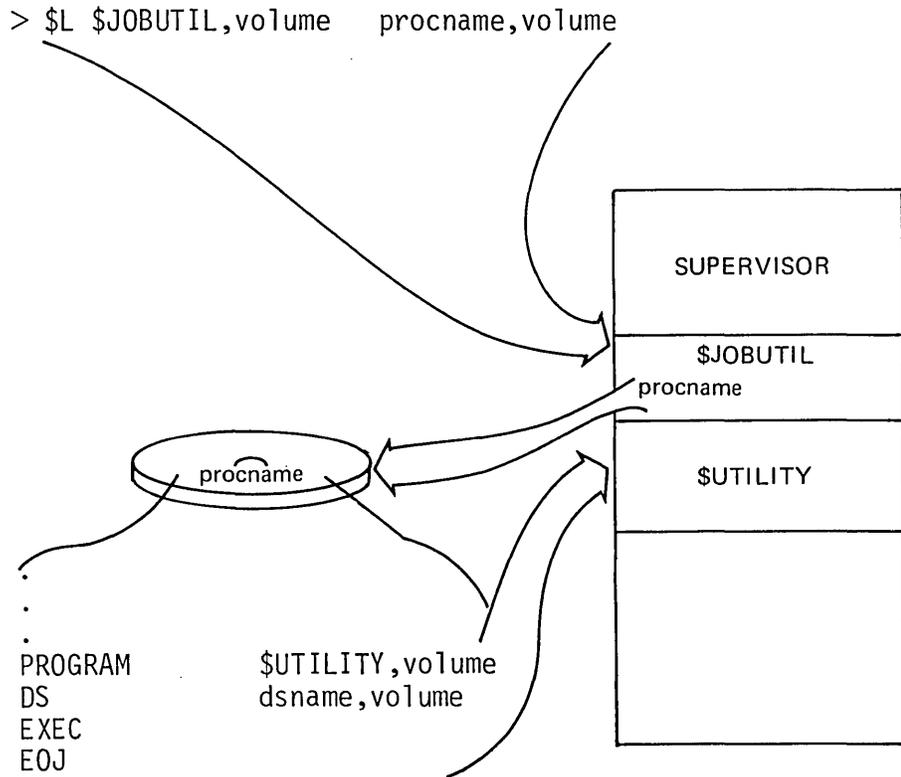


Figure 18-2. Loading utility using \$JOBUTIL

\$JOBUTIL, therefore, relieves the operator of the burden of having to remember and reenter each data set name and execution option with each reload of a utility or program. The data set names and options need only be entered once, into the job control procedure file. As long as the data sets and options described in the procedure file match the execution environment desired, all the operator needs to know is the name of the procedure file.

Although \$JOBUTIL provides obvious productivity advantages over the direct loading of programs and utilities using the \$L operator commands, a certain level of knowledge about the system must be attained before it can be used.

A thorough understanding of \$JOBUTIL operation, the meaning and organization of job procedure statements within a procedure file, and which programs or utilities might most profitably be run under \$JOBUTIL are necessary before procedure files can be created.

A procedure file, once created, is useful only so long as all data sets and operating parameters within that file exactly match the desired operation. If a single data set name or execution option is to be changed, the operator must

1. Use the \$L command to load the program or utility directly, in which case, not only the changed data set name or option, but all data set names and options must be entered
- or
2. Use \$FSEDIT or \$EDIT1N to make the required change in the procedure file

In summary, the \$L command is the most flexible facility for loading programs and utilities. At each load, the operator has the opportunity to change any or all options or data set names required by the program being loaded. The drawbacks of this method are that a large number of keystrokes is required, with the attendant possibility of operator input error; the operator must remember what may be a large number of data set names, all of which must be spelled correctly; and that all of the above is true each time the load is repeated, whether changes are required or not.

On the other hand, \$JOBUTIL, with its associated job control procedure files, is the most efficient means of loading programs and utilities, but lacks flexibility. If a change is required, the user must revert to the \$L facility, or edit a procedure file, and must have a fair amount of system experience to create the procedures initially.

The Session Manager is a productivity aid designed to take advantage of the efficiencies of \$JOBUTIL, without losing the flexibility of the \$L operator command.

The overall concept of Session Manager operation is illustrated in Figures 18-3 and 18-4. When a terminal user logs on (loads) the Session Manager, a menu of options is displayed on the screen. The operator enters the number associated with the option desired (Part A, Figure 18-3).

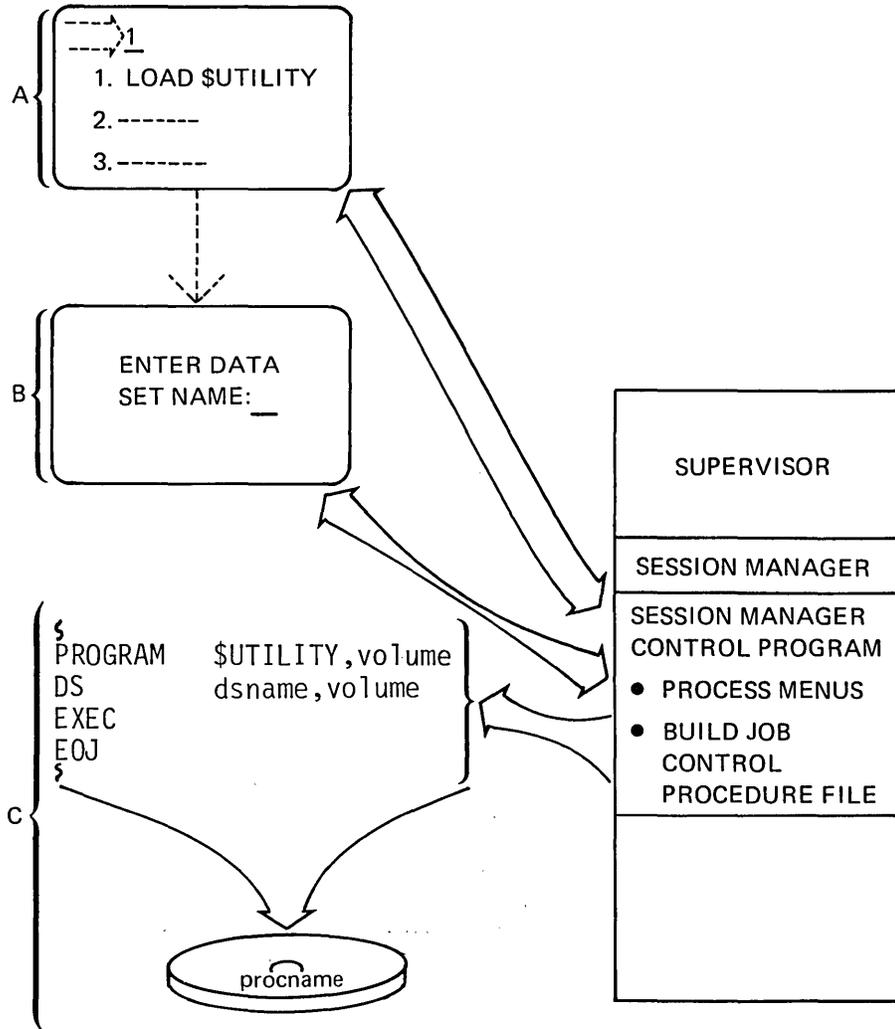


Figure 18-3. Session manager overview (1)

If the utility being loaded requires data set names or execution parameters, the Session Manager will display a parameter entry screen (Part B, Figure 18-3).

The Session Manager uses the operator input from the option and parameter entry screens to build a \$JOBUTIL job control procedure file (Part C, Figure 18-3).

After all inputs required to complete the procedure file have been entered, the Session Manager loads \$JOBUTIL, passes it the procedure file, and the program is loaded, and executed under control of the job stream processor (Figure 18-4).

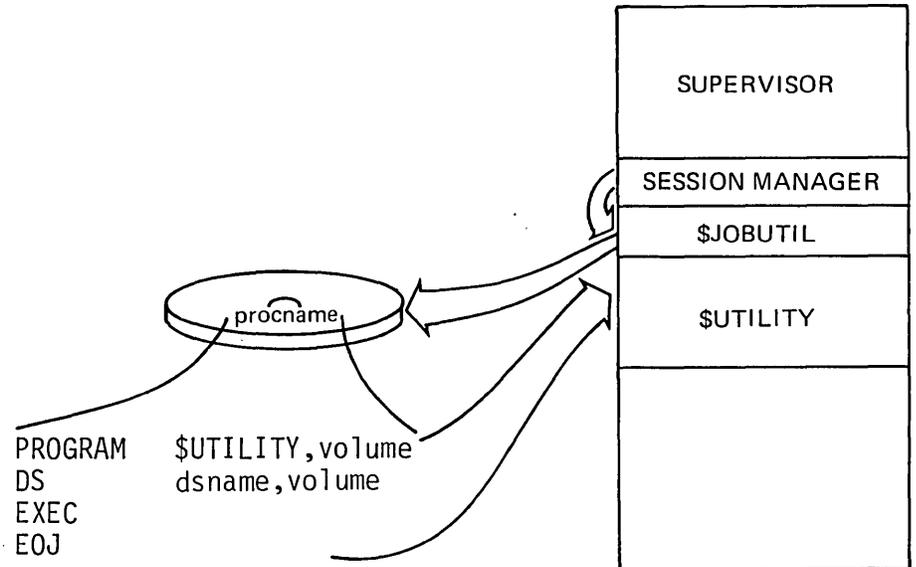


Figure 18-4. Session manager overview (2)

The next time the same option is chosen (Part A, Figure 18-3), the Session Manager will again display the parameter entry screen (Part B, Figure 18-3), allowing the operator to make changes, if desired. If no change from the previous execution is required, the operator just presses ENTER (null entry), and the Session Manager uses the parameters established in the previous execution again.

Without knowledge of job stream processor operation or job control procedure statements, the operator is able to take advantage of the efficiencies of running under \$JOBUTIL, while retaining the flexibility of easy alteration of execution parameters.

## SESSION MANAGER OPERATING CONCEPTS

READING ASSIGNMENT: IBM Series/1 Event Driven Executive Operator's Reference Message and Codes (SC34-1703), "Session Manager."

*Note:* The following discussion is intended to convey the general concepts of Session Manager organization and operation, not the actual detail. The menus, procedures, and decision tables shown in the illustrations are abbreviated and simplified for clarity.

### Definition of Terms

#### *Menus*

In discussing the Session Manager, the term "menu" is frequently used, usually in the context of "Primary Option Menu", "Secondary Option Menu", or "Parameter Selection Menu." A menu is nothing more than a full screen image on a 4978/4979/3101 M2 Display. Primary and secondary option menus usually consist of a numbered list of possible functions. To invoke a function, the operator enters the number of that option. For parameter selection menus, the operator fills in data entry areas with data set names, run parameters, etc., for the utility being invoked.

#### *Decision Tables*

Decision tables are associated with option menus, and are used by the Session Manager to decide what to do when a given option in an option menu is selected. For every option in an option menu, there is a corresponding entry in that menu's associated decision table. That decision table entry may direct the Session Manager to display another option menu, to display a parameter selection menu, or, if the option selected requires no further menus, to submit a procedure file to \$JOBUTIL for execution.

#### *Procedures*

A procedure is a \$JOBUTIL procedure file used by the Session Manager to execute programs/utilities selected by entry of menu options. Parameters and data set names required by some utilities are filled in with entries from parameter selection menus, and the procedure is passed to \$JOBUTIL and executed as a result of option menu decision table entries.

The relationship between option menus, option menu decision tables, parameter selection menus, and \$JOBUTIL procedures is illustrated in Figures 18-5 through 18-7. In Figure 18-5, the operator has selected option 1, Text Editing, from the primary option menu. The corresponding entry in the primary option decision table directs the Session Manager to pass a procedure control file to \$JOBUTIL which will load the full screen text editor, \$FSEDIT. No secondary option menu is necessary, because \$FSEDIT is the only text editor that the session manager supports, and therefore no choice of editors need be made.

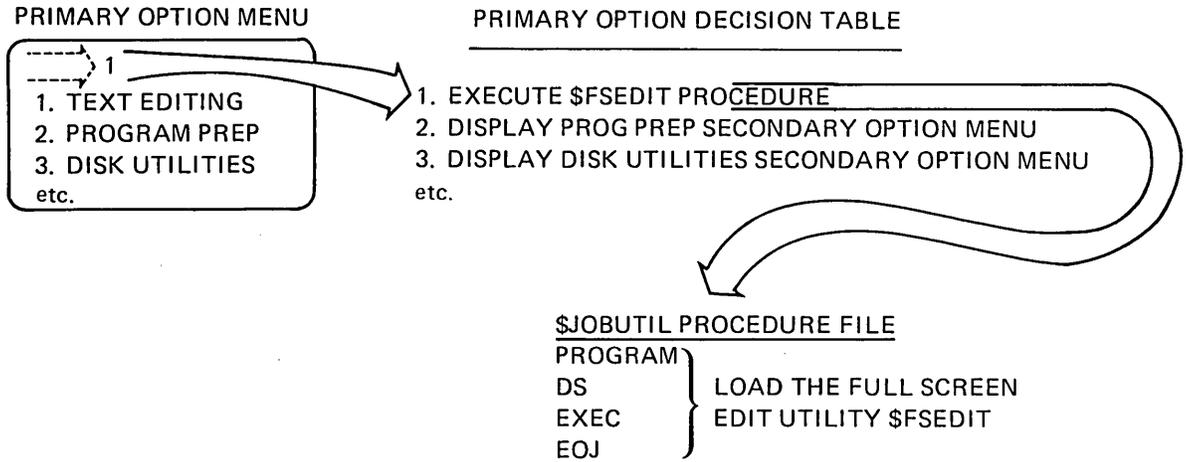


Figure 18-5. Session manager operation (1)

\$FSEDIT needs no parameters, so a parameter selection menu is not required.

*Note:* \$FSEDIT does require an edit work data set, but this data set is automatically allocated by the Session Manager when the terminal operator "logs on" to the session manager, and the name of the data set is automatically passed to the \$FSEDIT procedure control file, so the operator is not required to supply the edit work data set name when the Text Editing (option 1) function is requested.

In Figure 18-6, the operator has chosen option 3, Disk Utilities, on the primary option menu. Since there are several disk utility programs, the primary option decision table entry for option 3 directs the Session Manager to display a secondary option menu, so that the operator may choose which disk utility program to load.

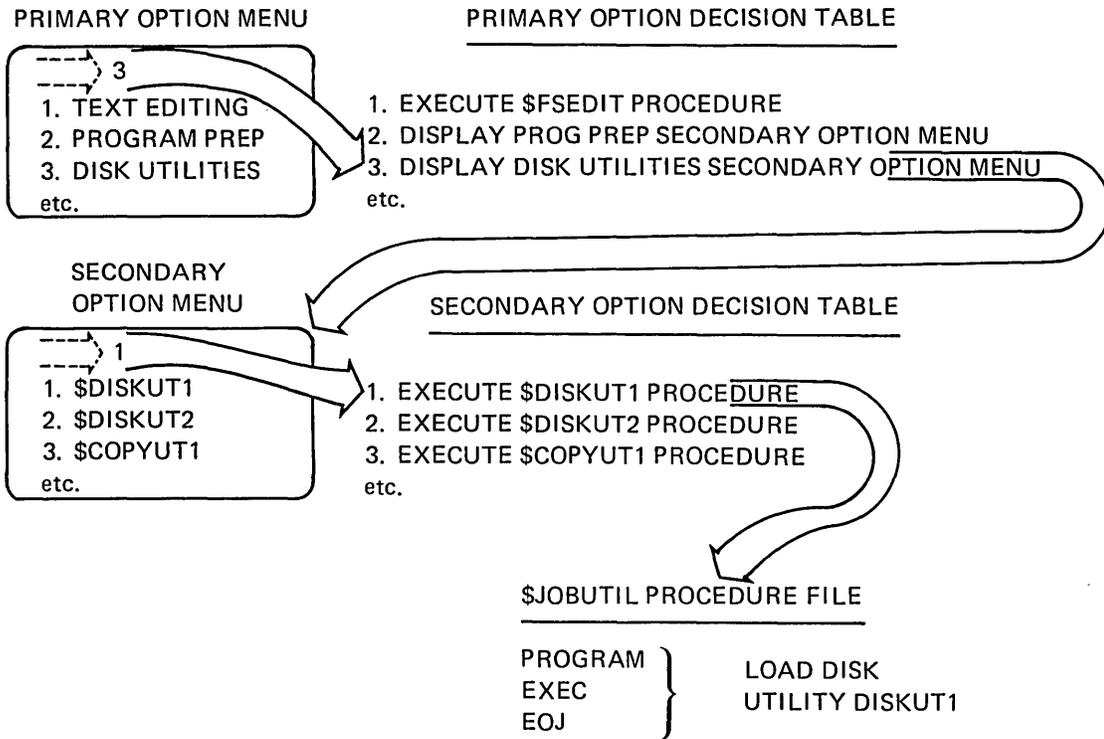


Figure 18-6. Session manager operation (2)

On the secondary option menu for the disk utilities, the operator has chosen option 1, \$DISKUT1. Because \$DISKUT1 requires no execution parameters or data set names, the secondary option menu decision table entry for option 1 directs the Session Manager to pass the \$DISKUT1 load procedure to \$JOBUTIL, which will result in the load of \$DISKUT1.

In Figure 18-7, the operator has chosen primary option 2, Program Preparation. Several different programs and utilities may be used in program preparation, so the primary option menu decision table entry directs the Session Manager to display the program preparation secondary option menu.

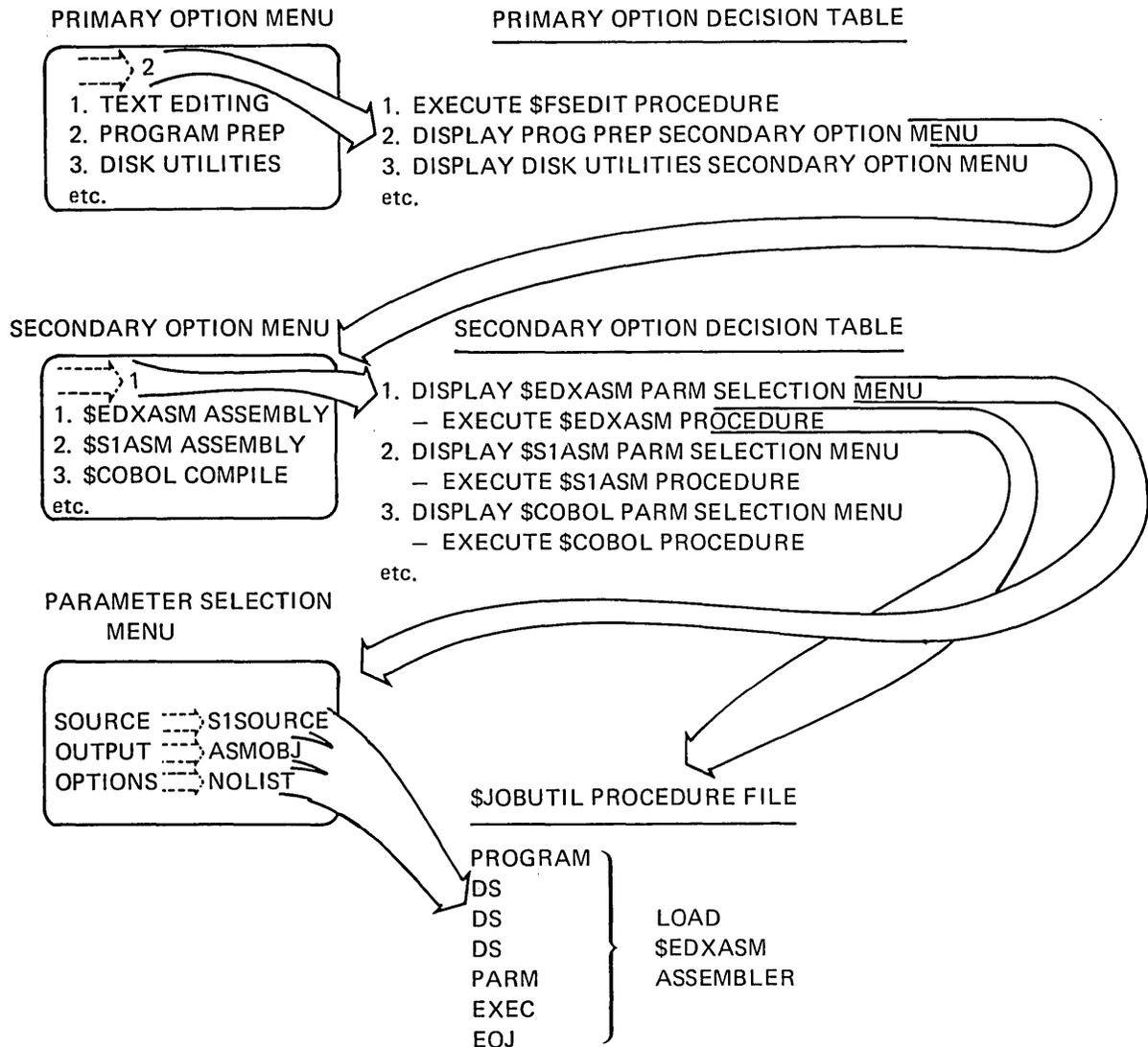


Figure 18-7. Session manager operation (3)

On the secondary option menu, the operator chooses option 1, \$EDXASM assembly. Since input and output data set names, as well as execution options may vary from one assembly to the next, the first part of the secondary option menu decision table entry for option 1 directs the Session Manager to display a parameter selection menu for \$EDXASM.

If no previous assembly has been done, the data set name and option entry areas on the screen will appear blank, and will be filled in at this time by the operator. If a previous assembly has been done, the data set names and assembler options used for the last assembly will be displayed. The operator may change items as necessary, or, if everything is the same as the previous assembly, may use all the same parameters, by pressing ENTER (null entry).

In any event, when the ENTER key is pressed, the Session Manager transfers the data from the parameter selection menu to the job control procedure. Then, under direction of the second part of the secondary option menu decision table entry for option 1, the procedure is passed to \$JOBUTIL for execution.

## USING THE SESSION MANAGER

The Session Manager is, to a large extent, self-tutoring, and is most easily learned by actually using it. However, in an attempt to convey at least an idea of what it is like to use the session manager, the three option selection sequences illustrated in Figures 18-5 through 18-7 will be repeated, this time using actual screen images that an operator would see while performing these operations.

## LOADING THE SESSION MANAGER

A 4978/4979/3101 M2 terminal is logically attached to the Session Manager by entering the \$L command shown in Figure 18-8.

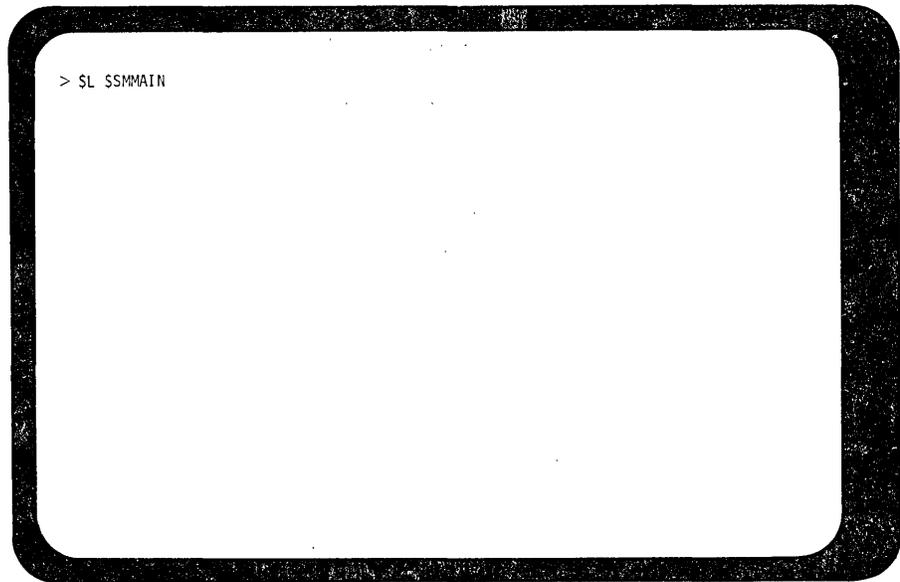


Figure 18-8. Session manager example (1)

*Note:* The Session Manager may be automatically brought up on all 4978/4979/3101 M2 terminals on the system, at IPL. See the reading assignment for details.

When the load command is honored, the LOGON screen in Figure 18-9 will appear. The session manager requires a unique 1 to 4 character user ID for each user. For this exercise, the characters XMPL are entered.

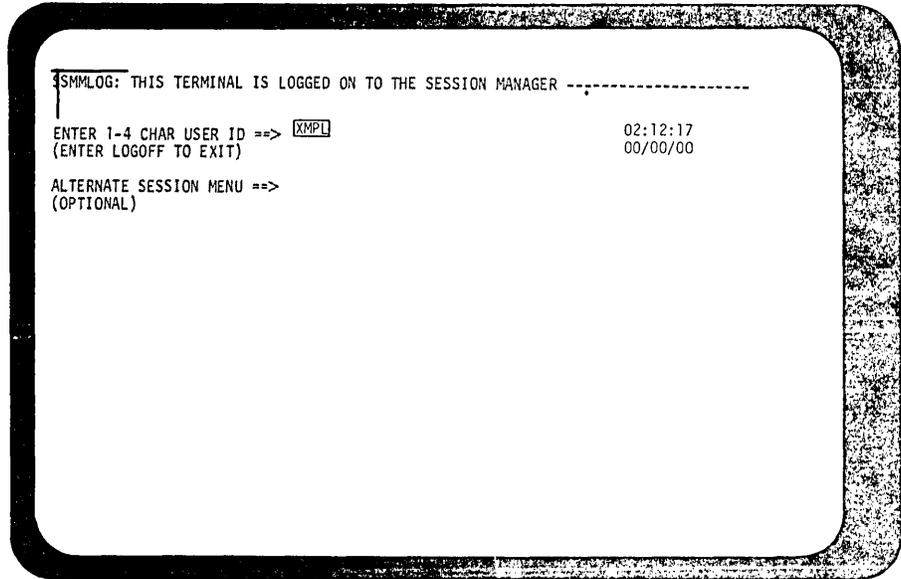


Figure 18-9. Session manager example (2)

The "ALTERNATE SESSION MENU" prompt below the user ID prompt would be used if you had created your own menus, decision tables, and procedures for use with the Session Manager.

## Data Set Allocation

After entering the user ID and pressing the ENTER key, the screen shown in Figure 18-10 appears.

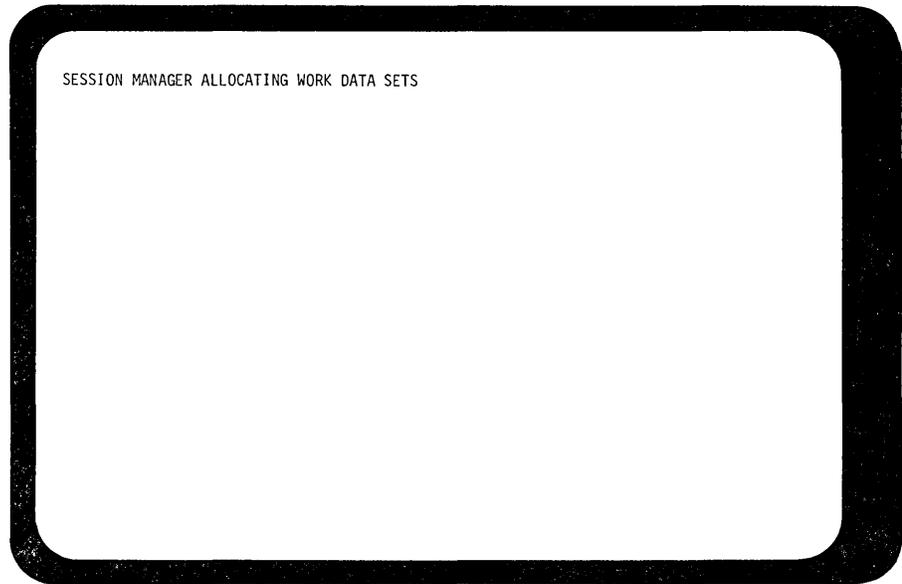


Figure 18-10. Session manager example (3)

When a user logs on to the Session Manager, the Session Manager allocates six data sets on EDX003. The names, sizes, and functions of these data sets are shown in Figure 18-11.

	SIZE (records)	
\$SMPxxxx	30	USED BY THE SESSION MANAGER TO SAVE PARAMETERS ENTERED FROM PARAMETER SELECTION MENUS DURING PREVIOUS SESSIONS UNDER SAME USER ID
\$SMWxxxx	30	USED BY SESSION MANAGER TO SUBMIT PROCEDURES TO \$JOBUTIL FOR EXECUTION

USED AS A WORK FILE FOR:

		\$FSE EDIT	\$EDX AS M	\$S1 AS M	\$L INK	\$C OB OL	\$F OR TR AN	\$P LI
\$SMExxxx	400	X						
\$SM1xxxx	400		X	X	X	X	X	X
\$SM2xxxx	400			X	X	X	X	X
\$SM3xxxx	250			X		X		X

Figure 18-11. Session manager data set allocation

The first four characters of each data set name is as depicted in Figure 18-11. The last 1 to 4 characters will be the user ID entered on the LOGON screen (Figure 18-9). In this case, the data sets would be named \$SMPXMPL, \$SMWXMPL, etc.

When attempting to allocate data sets, the Session Manager first checks to see if the data sets already exist, and if they do, will use those already there. If a user has allocated data sets (with the proper names) using \$DISKUT1, the user-allocated data sets will be used. This allows a user to define larger data sets than would the Session Manager, if the sizes allocated by the Session Manager prove too small.

After data sets have been allocated, the primary option menu in Figure 18-12 will appear.

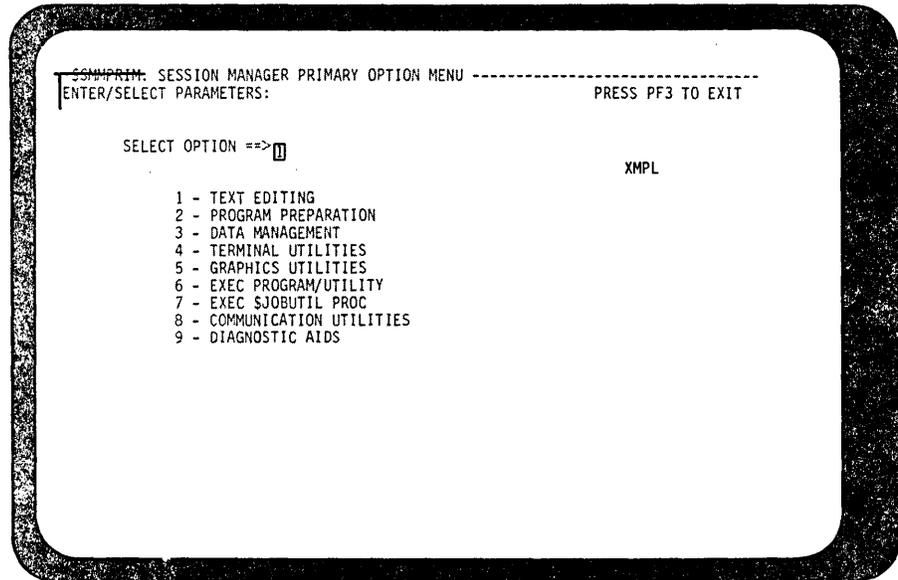


Figure 18-12. Session manager example (4)

The operator enters option 1 for TEXT EDITING, and the screen in Figure 18-13 appears.

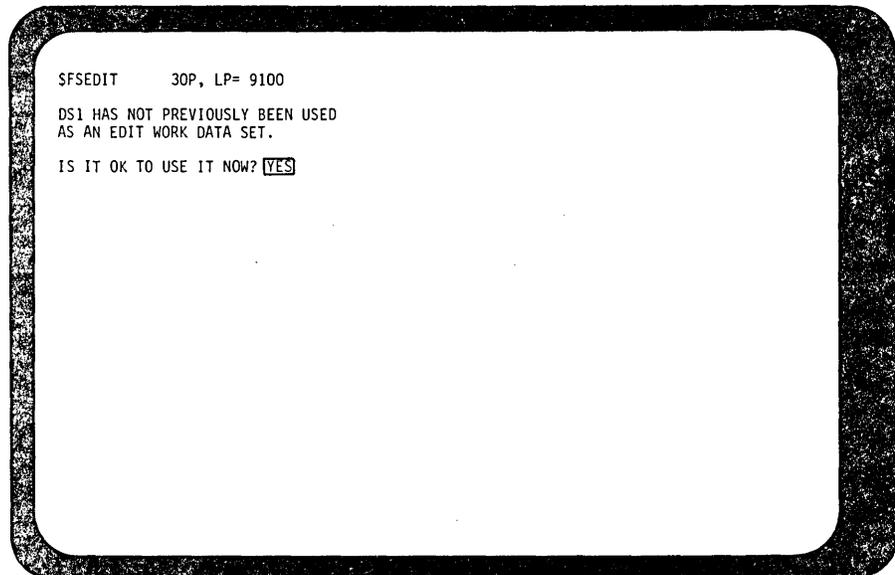


Figure 18-13. Session manager example (5)

The "IS IT OK TO USE IT NOW?" prompt appears because \$SMEXMPL, the text editor work data set, was just allocated, and the data is not in a format \$FSEDIT recognizes. After once being used for this purpose, the prompt will not reappear.

After responding YES to the prompt, the primary option menu for \$FSEDIT is displayed, just as it would if \$FSEDIT had been loaded using the \$L operator command (Figure 18-14).

```
----- SFSEDIT PRIMARY OPTION MENU -----  
SELECT OPTION ==>  
  
1 BROWSE - DISPLAY DATASET  
2 EDIT   - CREATE OR CHANGE DATASET  
3 READ   - READ DATASET FROM HOST/NATIVE  
4 WRITE  - WRITE DATASET TO HOST/NATIVE  
5 SUBMIT - SUBMIT BATCH JOB TO HOST SYSTEM  
6 LIST   - PRINT DATASET ON SYSTEM PRINTER  
7 MERGE  - MERGE DATA FROM A SOURCE DATASET  
8 END    - TERMINATE SFSEDIT  
9 HELP   - DISPLAY TUTORIAL
```

Figure 18-14. Session manager example (6)

When option 8, "TERMINATE \$FSEDIT" is entered, the screen in Figure 18-15 appears.

```
SFSEDIT ENDED AT 02:13:38  
SJOBUTIL ENDED AT 02:13:38  
DEPRESS ENTER KEY TO RETURN
```

Figure 18-15. Session manager example (7)

After a utility loaded by the Session Manager is ended, the ENTER key must be pressed to return to Session Manager control. Control is returned to the last Session Manager menu displayed before the utility was loaded, in this case, the Session Manager primary option menu (Figure 18-16).

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

SELECT OPTION ==> 3                                     XMPL

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

Figure 18-16. Session manager example (8)

This time, the operator enters option 3 for DISK UTILITIES, bringing up the data management utilities' secondary option menu in Figure 18-17.

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

SELECT OPTION ==> 1

1 - SDISKUT1 (DISK(ETTE) ALLOCATE, LIST DIRECTORY)
2 - SDISKUT2 (DISK(ETTE) DUMP/LIST DATASETS)
3 - SCOPYUT1 (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)

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.
```

Figure 18-17. Session manager example (9)

Entering option 1 on the secondary option menu results in the load of \$DISKUT1, as shown in Figure 18-18.

```
$DISKUT1 - DISK(ETTE) DATA MANAGEMENT UTILITY
*** JOB - $DISKUT1 - STARTED AT 02:14:49 00/00/00 ***

JOB      $DISKUT1 ($SMP0301)  USERID=XMPL

USING VOLUME EDX002

COMMAND (?): END

$DISKUT1 ENDED AT 02:15:00

$JOBUTIL ENDED AT 02:15:01

DEPRESS ENTER KEY TO RETURN
```

Figure 18-18. Session manager example (10)

When the utility is ended, and the ENTER key depressed, the Session Manager regains control, returning to the last screen displayed (Figure 18-19).

```
SSMM03: SESSION MANAGER DATA MANAGEMENT OPTION MENU-----
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

SELECT OPTION ==>

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)

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.
```

Figure 18-19. Session manager example (11)

To do Program Preparation, the operator now must return to the primary option menu. To return to a previous screen, press PF3 (Figure 18-20).

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

SELECT OPTION ==>

                                XMPL

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
```

Figure 18-20. Session manager example (12)

When option 2 is entered on the primary option menu, the program preparation secondary option menu is displayed (Figure 18-21).

```
SSMM02: SESSION MANAGER PROGRAM PREPARATION OPTION MENU -----
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

SELECT OPTION ==>

1 - SEDXASM COMPILER
2 - SSIASM ASSEMBLER
3 - SCOBOL COMPILER
4 - SFORT FORTRAN COMPILER
5 - SLINK LINKAGE EDITOR
6 - SUPDATE
7 - SUPDATEH (HOST)
8 - SPREFIND
9 - SEDXASM/SLINK/SUPDATE
10 - SPLI COMPILER/SLINK/SUPDATE
```

Figure 18-21. Session manager example (13)

The operator enters option 1, the \$EDXASM assembler, and the \$EDXASM parameter selection menu is displayed (Figure 18-22).

```

$SMP0201: SESSION MANAGER $EDXASM PARAMETER INPUT MENU-----
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

SOURCE INPUT (NAME,VOLUME) ==>

OBJECT OUTPUT (NAME,VOLUME) ==>

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

-----
AVAILABLE PARAMETERS:
NOLIST
LIST TERMINAL-NAME      (WHERE TERMINAL-NAME IS OPTIONAL)
ERRORS TERMINAL-NAME   (WHERE TERMINAL-NAME IS OPTIONAL)
CONTROL DATASET,VOLUME
OVERLAY (#)            (WHERE # IS NUMBER OF AREAS FROM 1 TO 6)

DEFAULT PARAMETERS:
LIST $SYSPRTR

```

Figure 18-22. Session manager example (14)

In Figure 18-22, the source input, object output, and optional parameter entry areas are blank. This indicates that this is the first time that \$EDXASM has been invoked under this user ID (XMPL). The Session Manager saves input parameters between executions of a program within a session, and across sessions of the same user ID (data set \$SMPxxx in Figure 18-11). If \$EDXASM had previously been used with user ID XMPL, the previously entered parameters would appear in Figure 18-22.

The operator enters the parameters as shown in Figure 18-23.

```
TSMM0201: SESSION MANAGER SEDXASM PARAMETER INPUT MENU-----
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

SOURCE INPUT (NAME,VOLUME) ==> SOURCE

OBJECT OUTPUT (NAME,VOLUME) ==> ASMOUT

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

-----
AVAILABLE PARAMETERS:
NOLIST
LIST TERMINAL-NAME      (WHERE TERMINAL-NAME IS OPTIONAL)
ERRORS TERMINAL-NAME   (WHERE TERMINAL-NAME IS OPTIONAL)
CONTROL DATASET,VOLUME
OVERLAY (#)            (WHERE # IS NUMBER OF AREAS FROM 1 TO 6)

DEFAULT PARAMETERS:
LIST $SYSRTR
```

Figure 18-23. Session manager example (15)

Before pressing the ENTER key, assume the operator notices that the output data set name is ASMOUT, when it should actually be ASMOBJ. The operator can, using the cursor movement keys, position the cursor so as to correct the erroneous spelling, or can press PF2, resulting in the screen in Figure 18-24.

```
TSMM0201: SESSION MANAGER SEDXASM PARAMETER INPUT MENU-----
ENTER/SELECT PARAMETERS:                                PRESS PF3 TO RETURN

SOURCE INPUT (NAME,VOLUME) ==>

OBJECT OUTPUT (NAME,VOLUME) ==>

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

-----
AVAILABLE PARAMETERS:
NOLIST
LIST TERMINAL-NAME      (WHERE TERMINAL-NAME IS OPTIONAL)
ERRORS TERMINAL-NAME   (WHERE TERMINAL-NAME IS OPTIONAL)
CONTROL DATASET,VOLUME
OVERLAY (#)            (WHERE # IS NUMBER OF AREAS FROM 1 TO 6)

DEFAULT PARAMETERS:
LIST $SYSRTR
```

Figure 18-24. Session manager example (16)

PF2 returns a menu to the state it was in when it was initially displayed, before operator entries were made to alter it. PF2 may be pressed any time before pressing ENTER, which signals completion of entry to a screen.

After reentering the parameters, this time with the correct spelling for the object output data set name, the \$EDXASM parameter selection menu looks like Figure 18-25.

```

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

                                                                02:17:20
                                                                00/00/00
                                                                XMPL

SOURCE INPUT (NAME,VOLUME) ==> SOURCE
OBJECT OUTPUT (NAME,VOLUME) ==> ASMOBJ
ENTER OPTIONAL PARAMETERS BY POSITION ==> NOLIST
                                                                1-----2-----
                                                                LIST   PRINTER NAME
                                                                NOLIST
                                                                ERRORS

                                                                DEFAULTS ARE: LIST  S$SYSPRTR

```

Figure 18-25. Session manager example (17)

Notice that the operator is not required to enter the name of a work data set. The Session Manager will supply the name of one of the work data sets that were automatically allocated during LOGON of the session.

Before pressing ENTER, assume the operator has mounted a diskette volume in the diskette device at address X'02', and wishes to bring the volume on line. This requires access to the system operator commands. Any time a Session Manager menu is displayed, the operator can get into system command mode by pressing PF1. When PF1 is pressed, the currently displayed menu is replaced with the screen shown in Figure 18-26.

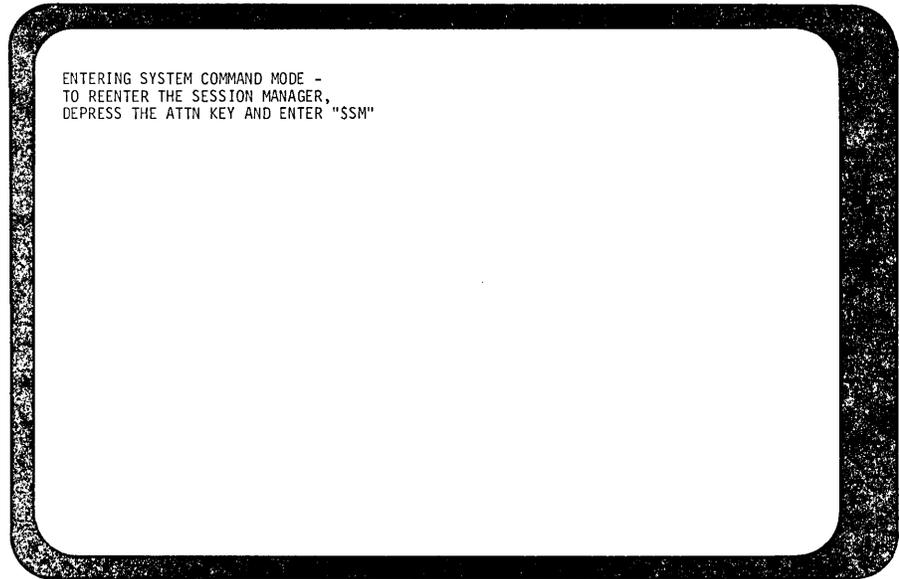


Figure 18-26. Session manager example (18)

The operator varies the volume on line, and enters \$SM to return to the Session Manager.

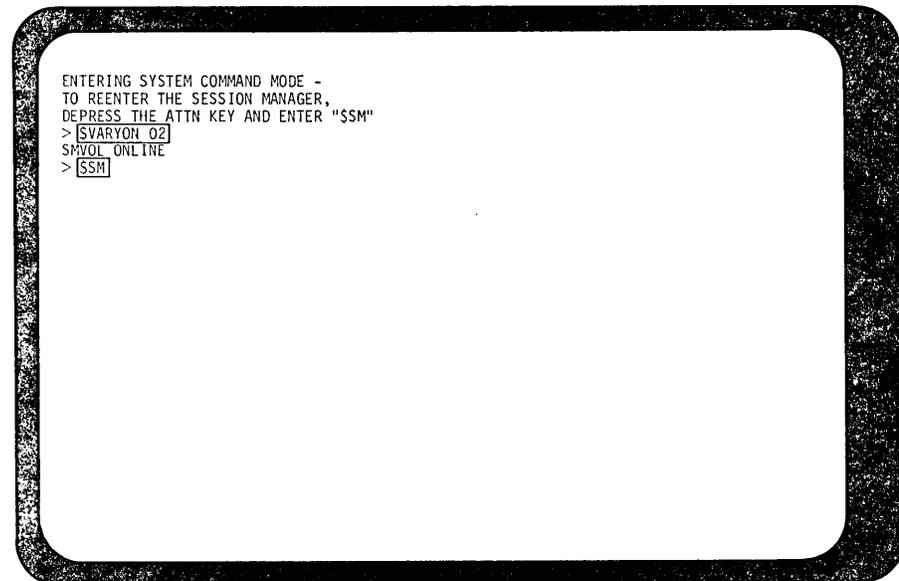


Figure 18-27. Session manager example (19)

When the ENTER key is pressed to enter the \$SM command, the Session Manager returns to the same menu that was displayed at the time system command mode was entered (PF1 was pressed).

```
SSM0201: SESSION MANAGER SEDXASM PARAMETER INPUT MENU-----
INTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

SOURCE INPUT (NAME,VOLUME) ==> SOURCE

OBJECT OUTPUT (NAME,VOLUME) ==> ASMOBJ

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

-----
AVAILABLE PARAMETERS:
NOLIST
LIST TERMINAL-NAME      (WHERE TERMINAL-NAME IS OPTIONAL)
ERRORS TERMINAL-NAME   (WHERE TERMINAL-NAME IS OPTIONAL)
CONTROL DATASET,VOLUME
OVERLAY (#)             (WHERE # IS NUMBER OF AREAS FROM 1 TO 6)

DEFAULT PARAMETERS:
LIST $$SYSPRTR
```

Figure 18-28. Session manager example (20)

Since all \$EDXASM parameters have been entered, the operator presses the ENTER key to submit the job for execution.

```
ASSEMBLE SOURCE TO ASMOBJ
*** JOB - SEDXASM - STARTED AT 02:18:14 00/00/00 ***

JOB      SEDXASM ($SMP0201)  USERID=XMPL
SEDXASM  70P,02:18:18, LP= 8A00

$JOBUTIL ENDED AT 02:18:51

DEPRESS ENTER KEY TO RETURN
```

Figure 18-29. Session manager example (21)

After the assembly is complete, the operator presses ENTER to return to the Session Manager, which brings up the \$EDXASM parameter selection menu, the last menu displayed.

```
SSMM021: SESSION MANAGER $EDXASM PARAMETER INPUT MENU-----
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

SOURCE INPUT (NAME,VOLUME) ==> SOURCE

OBJECT OUTPUT (NAME,VOLUME) ==> ASMOBJ

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

-----
AVAILABLE PARAMETERS:
NOLIST
LIST TERMINAL-NAME (WHERE TERMINAL-NAME IS OPTIONAL)
ERRORS TERMINAL-NAME (WHERE TERMINAL-NAME IS OPTIONAL)
CONTROL DATASET,VOLUME
OVERLAY (=) (WHERE = IS NUMBER OF AREAS FROM 1 TO 6)

DEFAULT PARAMETERS:
LIST SSYSPRTR
```

Figure 18-30. Session manager example (22)

Pressing PF3 twice backs out through the two previous screens, as shown in Figures 18-31 and 18-32.

```
SSMM02: SESSION MANAGER PROGRAM PREPARATION OPTION MENU-----
ENTER/SELECT PARAMETERS:                                     PRESS PF3 TO RETURN

SELECT OPTION ==>

1 - SEDXASM COMPILER
2 - SSIASM ASSEMBLER
3 - SCOBOL COMPILER
4 - SFORT FORTRAN COMPILER
5 - SLINK LINKAGE EDITOR
6 - SUPDATE
7 - SUPDATEH (HOST)
8 - SPREFIND
9 - SEDXASM/SLINK/SUPDATE
10 - SPLI COMPILER/SLINK/SUPDATE
```

Figure 18-31. Session manager example (23)

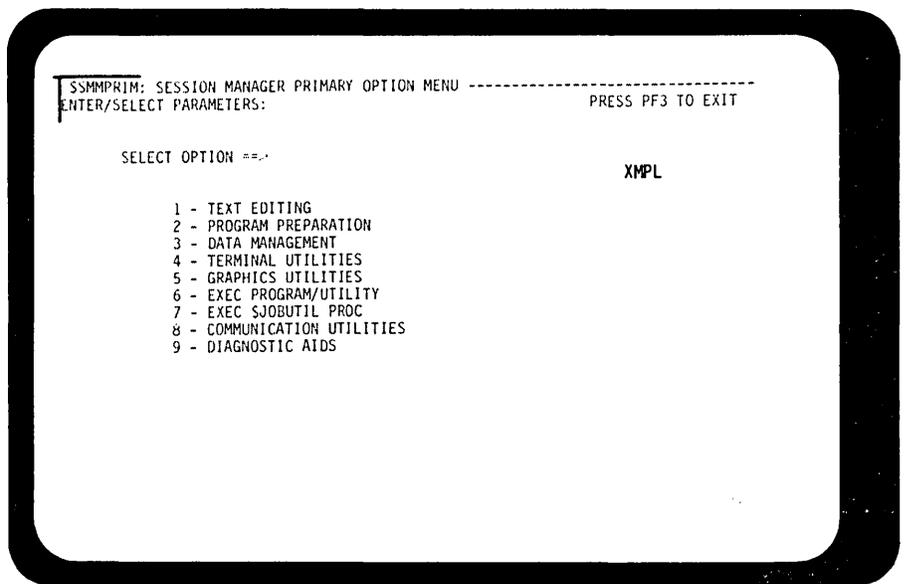


Figure 18-32. Session manager example (24)

To end the session, the operator again presses PF3, while the primary option menu is displayed, which results in the prompt in Figure 18-33.

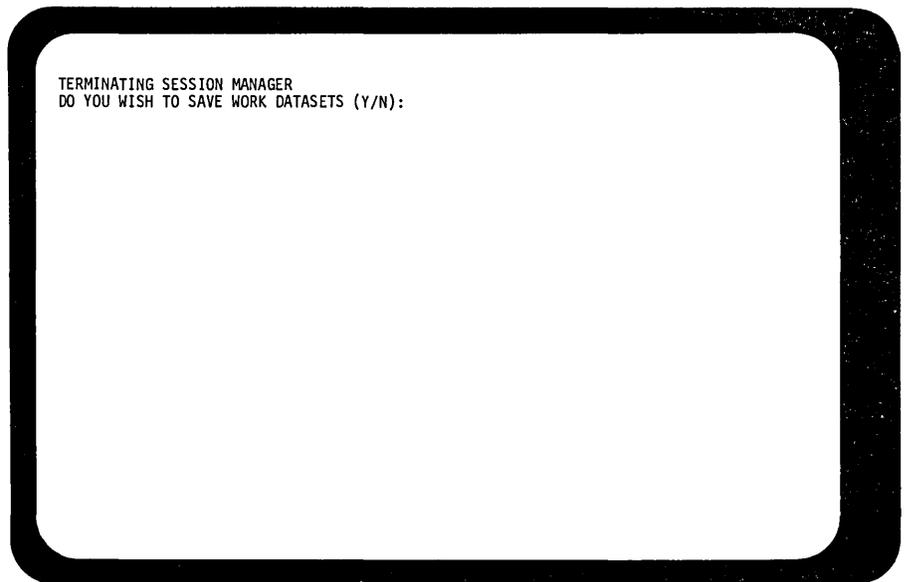


Figure 18-33. Session manager example (25)

If the operator replies YES to the prompt, none of the data sets allocated under this user ID at the beginning of the session will be deleted. If, for example, you had allocated work data sets larger than those normally allocated by the Session Manager (using \$DISKUT1), to accommodate particularly large assemblies or compiles, entering YES would prevent those data sets from being deleted, and the next time you wished to log on with the same ID, you would not have to first use \$DISKUT1 to allocate your oversize work files.

Entering NO results in deletion of all the data sets under this ID except for \$SMPxxxx. This data set is retained, and used to save parameters for future sessions under the same ID. For example, if the operator were ever again to log on with ID XMPL, and through the option menus, choose program preparation and \$EDXASM assembly, when the \$EDXASM parameter selection menu was displayed, the parameters would be those last entered during this session.

Assuming NO was entered, the message in Figure 18-34 would be displayed during deletion of the data sets, followed by the LOGON screen in Figure 18-35.

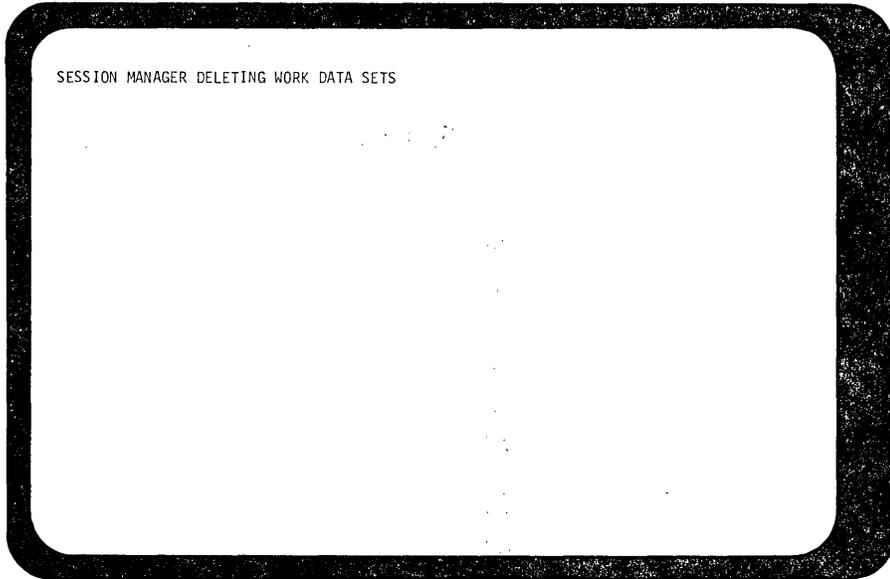


Figure 18-34. Session manager example (26)

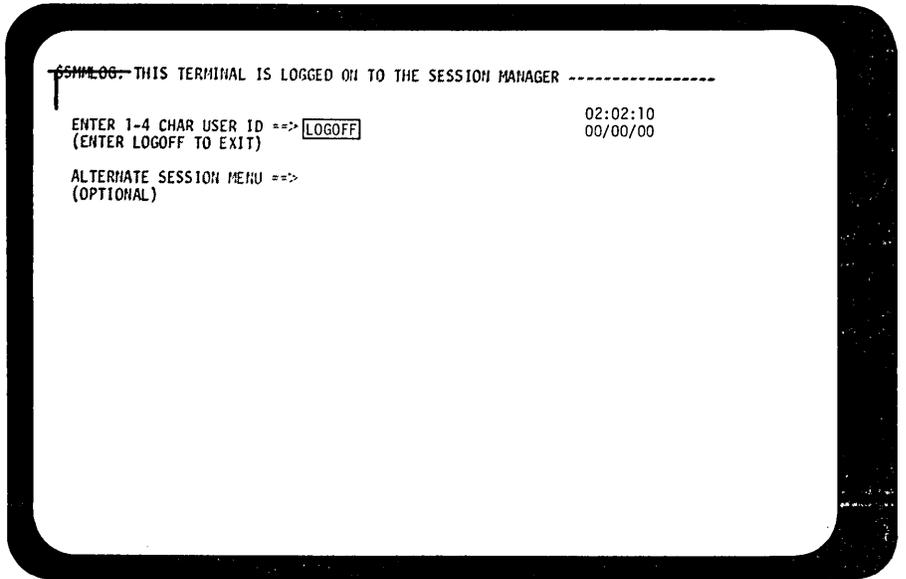


Figure 18-35. Session manager example (27)

To terminate the session, the operator enters LOGOFF in the command input area, and presses ENTER (Figure 18-36).

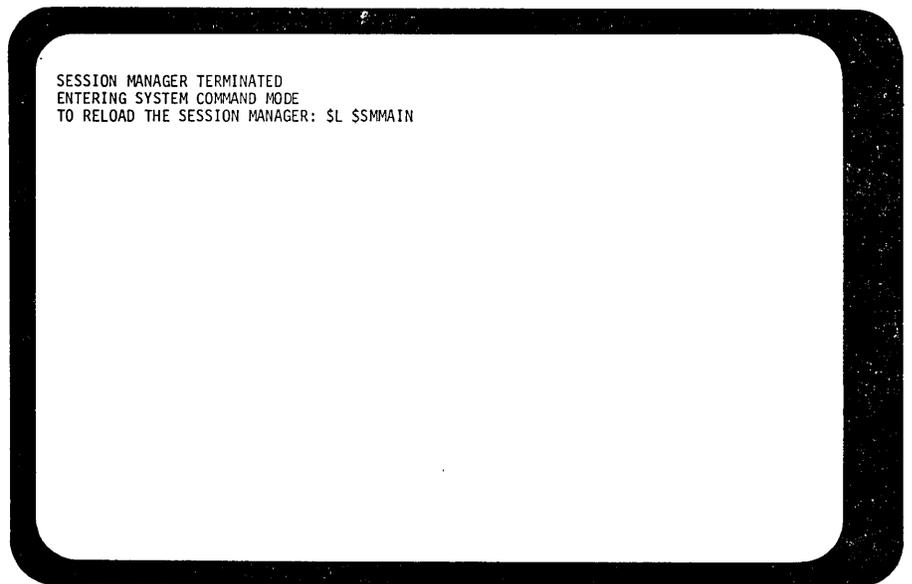


Figure 18-36. Session manager example (28)

**This page intentionally left blank.**

# Appendix A. SYSGEN Listings

```

LOG          $SYSPRTR
*** JOB - $SUPPREP - STARTED AT 03:22:45 00/00/00 ***

JOB          $SUPPREP
PROGRAM     $EDXASM,ASMLIB
NOMSG
PARM
DS          $EDXDEFS,EDX002
DS          ASMWORK,EDX002
DS          ASMOBJ,EDX002
EXEC
  
```

Figure A-1. Procedure file statements controlling assembly

```

EDX ASSEMBLER STATISTICS

SOURCE INPUT - $EDXDEFS,EDX002
WORK DATA SET - ASMWORK ,EDX002
OBJECT MODULE - ASMOBJ ,EDX002
DATE: 00/00/00 AT 03:23:32
ASSEMBLY TIME: 35 SECONDS
STATEMENTS PROCESSED - 30

NO STATEMENTS FLAGGED

LOC  +0  +2  +4  +6  +8          SOURCE STATEMENT          $EDXDEFS,EDX002  (5719-XX4)-V3.0.0  0/00/ 0  3:23

0000          $EDXDEF  CSECT          00000010
0000  0000          DATA  F*0'          00000020
*          00000030
* EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0  00000040
*          00000050
* THE FOLLOWING DEFINES THE STARTER SUPERVISOR AS SHIPPED ON THE  00000060
* DISKETTE LABELED XS3001. FOR COMPLETE DESCRIPTIONS OF THESE  00000070
* STATEMENTS OR ANY OTHER SYSTEM DEFINITION STATEMENTS, REFER TO  00000080
* THE EDX VERSION 3 SYSTEM GUIDE: SC34-1702  00000090
*          00000100
*          00000110

0002  0000  0000  0000  0000  0000          SYSTEM STORAGE=(10,10,10),PARTS=(8,16,20)
0052  000A  000A  000A  0000  0000
005C  0000  0000  0000  0020  FFFF
0066  0008  0010  0014  0000  0000
0070  0000  0000  0000  0098  00EC
007A  0140  0194  0198  019C  01A0
0084  01A4  00E8  013C  0190  0194
008E  0198  019C  01A0  01A4  0000
0098  FFFF  3000  FFFF  8000  FFFF
00A2  8000  FFFF  8000  FFFF  8000
00AC  FFFF  3000  FFFF  8000  FFFF
00B6  8000  FFFF  8000  FFFF  8000
00C0  FFFF  8000  FFFF  8000  FFFF
00CA  8000  FFFF  8000  FFFF  8000
00D4  FFFF  8000  FFFF  8000  FFFF
00DE  8000  FFFF  8000  FFFF  8000
00E8  FFFF  8000  FFFF  8000  FFFF
00F2  8000  FFFF  8000  FFFF  8000
00FC  FFFF  8000  FFFF  8000  FFFF
0106  8000  FFFF  8000  FFFF  8000
0110  FFFF  8000  FFFF  8000  FFFF
011A  8000  FFFF  8000  FFFF  8000
0124  FFFF  8000  FFFF  8000  FFFF
012E  8000  FFFF  8000  FFFF  8000
0138  FFFF  8000  FFFF  8000  FFFF
0142  8000  FFFF  8000  FFFF  8000
014C  FFFF  3000  FFFF  8000  FFFF
0156  8000  FFFF  8000  FFFF  8000
0160  FFFF  8000  FFFF  8000  FFFF
016A  8000  FFFF  8000  FFFF  8000
0174  FFFF  3000  FFFF  8000  FFFF
017E  8000  FFFF  8000  FFFF  8000
018B  FFFF  8000  FFFF  8000  FFFF
  
```

Figure A-2. Assembly statistics and listing (1 of 6)

0192	8000	FFFF	8000	FFFF	8000			
019C	FFFF	8000	FFFF	8000	FFFF			
01A6	8000	0003	0000	FFFF				
01AF	0000	0028	2440	0000	0000	TIMER	ADDRESS=40	00000111
01B8	0000	0000	0000	0000	0000			
01C2	0000	0000	0000	03E8	4E1F			
01CC	1F1C	1F1E	1F1E	1F1F	1E1F			
01D6	1E1F	0000	0028	2441	0000			
01E0	6E41	0000	6441	0000	6741			
01EA	0000	01A8	0000	0000	0000			
01F4	088E	EA60						
01F8	0800	0000	58C4	C4C5	F0F3	DISK	DEVICE=4962-3,ADDRESS=03	00000120
0202	0000	0169	0000	02F0	FFFF			
020C	0000	0000	0000	0000	02DE			
0216	0226	0003	003C	0000	0000			
0220	0000	0000	0000	0000	00CA			
022A	0000	0400	0000	01F8	030C			
0234	0000	0000	03CC	0000	0000			
023E	0000	0000	0000	0000	0000			
0252	0000	0000	7003	0001	8007			
025C	0000	0000	0000	0000	026A			
0266	0000	0000	8005	0000	0000			
0270	0000	0000	027A	0000	0000			
027A	2009	0000	0000	0000	0000			
0284	028A	0000	0000	0000	0000			
028E	0000	0000	0000	0000	0000			
0298	0000	7F03	029E	2000	0000			
02A2	0000	0000	0000	0000	0008			
02AC	02AE	0000	0000	0000	0000			
02B6	0000	0000	0000	0000	0000			
02DE	0000	0000	0000	0000	0000	DISK	DEVICE=4964,ADDRESS=02,TASK=YES,END=YES	00000130
02E8	0000	0000	0000	0000	FFFF			
02F2	0000	0000	0000	0000	0000			
02FC	030C	0000	0000	0000	0000			
0306	0000	0000	0000	0000	0106			
0310	0000	0000	0000	020E	0000			
031A	0000	0000	044C	0000	0000			
0324	0000	0000	0000	0000	0000			
0338	0000	0000	7002	0001	8007			
0342	0000	0000	0000	0000	0350			
034C	0000	0000	8005	0000	0000			
0356	0000	0000	0360	0000	0000			
0360	2009	0000	0000	0000	0000			
036A	0370	0000	0000	0000	0000			
0374	0000	0000	0000	0000	0000			
037E	0000	7F02	0384	2000	0000			
0388	0000	0000	0000	0000	0008			
0392	0394	0000	0000	0000	0000			
039C	0000	0000	0000	0000	0000			
03C4	0606	4040	4040	4040	0000			
03CE	0000	0000	0000	0000	0000			
03D8	0000	03DA	03CC	0000	0000			
03E2	0000	0000	0000	0000	0001			
03EC	0096	0000	0000	FFFF	0000			
03F6	0000	03F8	0000	0000	03FA			
0400	58C4	E3E2	02F1	4040	0000			
040A	0000	0000	0000	0000	0000			
0414	0000	FFFF	0000	0000	0000			
041E	0000	0000	03CC	0000	0000			
0428	0000	0000	0000	0000	0000			
0446	0000	03CC	0080	0000	0000			
0450	0000	0000	0000	0000	0000			
045A	045A	044C	0000	0000	0000			
0464	0000	0000	0000	0001	0096			
046E	0000	0000	FFFF	0000	0000			
0478	0478	0000	0000	047A	58C4			
0482	F3E2	02F2	4040	0000	0000			
048C	0000	0000	0000	0000	0000			
0496	FFFF	0000	0000	0000	0000			
04A0	0000	044C	0000	0000	0000			
04AA	0000	0000	0000	0000	0000			
04CB	044C	0080						
04CC	052E	0000	0000	0000	0000	\$SYSLOG	TERMINAL	DEVICE=4979,ADDRESS=04,HDCOPY=\$SYSPRTR,PART=2
04D6	0000	0000	0000	0000	0000			00000150
04E0	0000	0000	6004	0003	6F04			
04EA	0000	2004	0406	7004	04D4			
04F4	7F04	0404	0000	0000	0006			
04FE	0400	0000	0000	0000	0017			
0508	0018	0050	0C00	0050	000C			
0512	0017	0018	0050	0C00	0050			
051C	1850	5BE2	E8E2	D3D6	C740			
0526	0000	FFFF	0000	0708	0000			
0530	0406	0000	0000	FFFF	0000			
053A	6E6D	0020	6F03	0000	0000			
0544	0582	FFFF	4324	052E	6302			
054E	054C	5000	C3A5	C072	6F03			
0558	0000	0000	0582	634D	0072			
0562	4324	052E	5600	6E6D	004E			
056C	6F03	0000	0000	0628	0000			

Figure A-2. Assembly statistics and listing (2 of 6)

```

0576 4324 052E 6802 057A 0628
0580 0FAC 0000 0000 0000 FFFF
058A 0000 0000 0000 0000 0000
0594 0000 0000 0100 0100 0000
059E 0000 0000 0000 0000 0000
05C6 0000 0000 0000 0000 05D6
05D0 05D6 0000 0000 0000 0000
05DA 0000 0000 0000 0000 0000
062A 0000 0000 0000 0000 0000
0634 0000 0000 0629 0000 0000
063E 0000 0000 0000 0000 0001
0648 000A 0000 0000 FFFF 0000
0652 0000 0654 0000 0000 0656
065C 02C2 E3C1 E2D2 0065 0000
0666 0000 0000 0000 0000 0000
0670 0000 FFFF 0000 0000 052E
067A 0000 0000 0628 0000 0000
0684 0000 0000 0000 0000 0000
0698 0000 0001 0000 0000 0000
06A2 0000 0628 0000
06A8 0000 0000 0000 0000 0000
06B0 0000 6006 0003 6F06 0000
06C6 2006 0406 7006 06AE 7F06
06D0 06AE 0000 0000 0006 0400
06DA 0000 0000 0000 0017 0018
06E4 0050 0C00 0050 0000 0017
06EE 0018 0050 0C00 0050 1850
06F8 C4E2 07D3 E8F1 4040 0000
0702 FFFF 0000 08CC 0000 040E
070C 0000 0000 FFFF 0000 6E6D
0716 0020 6F03 0000 0000 075C
0720 FFFF 4324 0709 6802 0726
072A 5000 C8A5 0072 6F03 0000
0734 0000 075C 6E6D 0072 4324
073E 0708 5600 6E6D 004E 6F03
0748 0000 0000 0802 0000 4324
0752 0708 6802 0754 0802 0FAC
075C 0000 0000 0000 FFFF 0000
0766 0000 0000 0000 0000 0000
0770 0000 0100 0100 0000 0000
077A 0000 0000 0000 0000 0000
07A2 0000 0000 0000 0780 0780
07AC 0000 0000 0000 0000 0000
0806 0000 0000 0000 0000 0000
0810 0000 0802 0000 0000 0000
081A 0000 0000 0000 0001 000A
0824 0000 0000 FFFF 0000 0000
082E 082E 0000 0000 0830 02C2
0838 F3C1 E2D2 0066 0000 0000
0842 0000 0000 0000 0000 0000
084C FFFF 0000 0000 0708 0000
0856 0000 0802 0000 0000 0000
0860 0000 0000 0000 0000 0000
0874 0002 0000 0000 0000 0000
087E 0802 0080
0882 6000 0003 6F00 0000 2000
088C 0000 5000 0000 1000 0000
0896 7F03 000A 0600 0400 0000
08A0 0000 0000 0017 0018 0050
08AA 0000 0050 0000 0017 0018
08B4 0050 0000 0050 4250 58E2
083E E3E2 03D6 C7C1 0000 FFFF
08C8 0000 0ADE 0000 0010 0000
08D2 0000 FFFF 0000 6E6D 0020
08DC 6F03 0000 0000 0920 FFFF
08E6 4324 08CC 6802 08EA 5000
08F0 C8A5 0072 6F03 0000 0000
08FA 0920 6B4D 0072 4324 03CC
0904 5600 6E6D 004E 6F03 0000
090E 0000 09C6 0000 4324 03CC
0918 6802 0916 09C6 0000 0000
0922 0000 0000 FFFF 0000 0000
092C 0000 0000 0000 0000 0000
0936 1800 0000 0000 0000 0000
0940 0000 0000 0000 0000 0000
0968 0000 0000 0974 0974 0000
0972 0000 0000 0000 0000 0000
09CC 0000 0000 0000 0000 0000
09D6 09C6 0000 0000 0000 0000
09E0 0000 0000 0001 000A 0000
09EA 0000 FFFF 0000 0000 09F2
09F4 0000 0000 09F4 02C2 E3C1
09FE E2D2 0067 0000 0000 0000
0A08 0000 0000 0000 0000 FFFF
0A12 0000 0000 08CC 0000 0000
0A1C 0000 0000 0000 0000 0000
0A26 0000 0000 0000 0000 0000
0A3A 0000 0000 0000 0000 09C6
DSPLY1 TERMINAL DEVICE=4979,ADDRESS=06,HOCOPY=$SYSPRTR,PART=3 00000151
$SYSL0GA TERMINAL DEVICE=TTY,ADDRESS=00,CRODELAY=4,PAGSIZE=24, 00000160
BOTM=23,SCREEN=YES 00000170

```

Figure A-2. Assembly statistics and listing (3 of 6)

```

0A44 0080
0A46 0000 0000 0000 0000 0000 $SYSLOGB TERMINAL DEVICE=ACCA,ADDRESS=08,MODE=31018,ADAPTER=SINGLE, C00000180
0A50 0000 0000 0000 0000 C400 0000 BITRATE=9600,RANGE=HIGH 00000190
0A5A 0000 0000 0000 7008 0A64
0A64 2004 0000 0000 0000 0000
0A6E 0000 0002 0386 0002 0002
0A78 000A 0002 0000 0000 0000
0A82 0000 0010 07FE 6080 8080
0A8C 8080 0000 0000 0000 6008
0A96 0003 6F08 0000 2008 0000
0AA0 7008 0A84 7F08 0A84 FE10
0AAA E050 0200 0440 0007 0000
0A34 0000 0017 0018 0050 0000
0ABE 00C8 0000 0017 0018 0050
0AC8 0000 00C8 1850 58E2 E8E2
0AD2 03D6 C7C2 0000 FFFF 0000
0ADC 0D94 0000 100E 0000 0000
0AE6 FFFF 0000 6E6D 0020 6F03
0AF0 0000 0000 0B32 FFFF 4324
0AFA 0ADE 6802 0AFC 5000 CBA5
0B04 0072 6F03 0000 0000 0B32
0B0E 684D 0072 4324 0ADE 5600
0B18 6E6D 004E 6F03 0000 0000
0B22 0CAA 0000 4324 0ADE 6802
0B2C 032A 0CAA 0000 0000 0000
0B36 0000 FFFF 0000 0000 0000
0B40 0000 0000 0000 0000 0816
0B4A 0886 0000 0000 0000 0000
0B54 0000 0000 0000 0000 0000
0B7C 0000 0886 0886 0000 0000
0B86 0000 0000 0000 0000 0000
0CB2 0000 0000 0000 0000 0CAA
0C8C 0000 0000 0000 0000 0000
0CC6 0000 0001 000A 0000 0000
0CD0 FFFF 0000 0000 0C06 0000
0CDA 0000 0C08 02C2 E3C1 E2D2
0CE4 0068 0000 0000 0000 0000
0CEE 0000 0000 0000 FFFF 0000
0CF8 0000 0ADE 0000 0000 0CAA
0D02 0000 0000 0000 0000 0000
0D20 0000 0000 0000 0CAA 0080
0D2A 0000 0000 0000 0000 0000 LINEPRTR TERMINAL DEVICE=4973,ADDRESS=21 00000250
0D34 0000 0000 0000 0080 0000
0D3E 0000 0000 0000 0000 0000
0D48 0000 6021 0003 6F21 0000
0D52 2021 0206 7021 0D3A 7F21
0D5C 0D3A 0000 0000 0000 0020
0D66 0000 0000 0003 033E 0042
0D70 0084 0300 0084 0003 033E
0D7A 0042 0084 0300 0084 FF84
0D84 03C9 05C5 0709 E3D9 0000
0D8E FFFF 0000 0FAC 0000 0306
0D98 0000 0000 FFFF 0000 6E6D
0DA2 0020 6F03 0000 0000 0DE8
0DAC FFFF 4324 0094 6802 0DB2
0D56 5000 CBA5 0072 6F03 0000
0DC0 0000 0DE8 6840 0072 4324
0DCA 0D94 5600 6E6D 004E 6F03
0DD4 0000 0000 0EC2 0000 4324
0DDE 0D94 6802 0DE0 0EC2 0000
0DE8 0000 0000 0000 FFFF 0000
0DF2 0000 0000 0000 0000 0000
0DFC 0000 0100 0000 0000 0000
0E06 0000 0000 0000 0000 0000
0E2E 0000 0000 0000 0E3C 0E3C
0E38 0000 0000 0000 0000 0000
0EC4 0000 0000 0000 0000 0000
0ECE 0000 0000 0EC2 0000 0000
0ED8 0000 0000 0000 0000 0001
0EE2 000A 0000 0000 FFFF 0000
0EEC 0000 0EEE 0000 0000 0EF0
0EF6 02C2 E3C1 E2D2 0069 0000
0F00 0000 0000 0000 0C00 0000
0FOA 0000 FFFF 0000 0000 0D94
0F14 0000 0000 0EC2 0000 0000
0F1E 0000 0000 0000 0000 0000
0F3C 0000 0EC2 0080
0F42 0000 0000 0000 0000 0000 $SYSRTR TERMINAL DEVICE=4974,ADDRESS=01,END=YES 00000251
0F4C 0000 0000 0000 0080 0000
0F56 0000 0000 0000 0000 0000
0F60 0000 6001 0003 6F01 0000
0F6A 2001 0206 7001 0F52 7F01
0F74 0F52 0000 0000 0000 0020
0F7E 0000 0000 0003 033E 0042
0F88 0084 0300 0084 0003 033E
0F92 0042 0084 0300 0084 FF84
0F9C 58E2 E8E2 0709 E3D9 0000
0FA6 FFFF 0000 0000 0000 0206
0F80 0000 0000 FFFF 0000 6E6D

```

Figure A-2. Assembly statistics and listing (4 of 6)

```

0FBA 0020 6F03 0000 0000 1000
0FC4 FFFF 4324 0FAC 6802 0FCA
0FCE 5000 CBA5 0072 6F03 0000
0FD8 0000 1000 6840 0072 4324
0FE2 0FAC 5600 6E60 004E 6F03
0FEC 0000 0000 10DA 0000 4324
0FF6 0FAC 6802 0FF8 10DA 0000
1000 0000 0000 0000 FFFF 0000
100A 0000 0000 0000 0000 0000
1014 0000 0100 0000 0000 0000
101E 0000 0000 0000 0000 0000
1046 0000 0000 0000 1054 1054
1050 0000 0000 0000 0000 0000
10DC 0000 0000 0000 0000 0000
10E6 0000 0000 10DA 0000 0000
10F0 0000 0000 0000 0000 0001
10FA 000A 0000 0000 FFFF 0000
1104 0000 1106 0000 0000 1108
110E D2C2 E3C1 E2D2 006A 0000
1118 0000 0000 0000 0000 0000
1122 0000 FFFF 0000 0000 0FAC
112C 0000 0000 10DA 0000 0000
1136 0000 0000 0000 0000 0000
1154 0000 10DA 0080
115A                                     $SYSCOM CSECT
115A FFFF 0000 0000 0000 0000 QCB
1164 FFFF 0000 0000 0000 0000 QCB
116E FFFF 0000 0000 ECB
1174 FFFF 0000 0000 ECB
117A                                     ENTRY $EDXPTCH
117A C000 0000 0000 0000 0000 $EDXPTCH DATA 128F'0'
1274 0000 0000 0000
127A                                     SYSTEM PATCH AREA

                                     END
                                     00000260
                                     00000270
                                     00000280
                                     00000290
                                     00000300
                                     00000310
                                     00000320
                                     00000330

EXTERNAL/UNDEFINED SYMBOLS

$EDXDEF ENTRY 0000
SVC EXTRN
SUPEXIT EXTRN
SETBUSY EXTRN
RETURN EXTRN
SVC1 WXTRN
POST WXTRN
$SVCIBUF ENTRY 0002
$VCBF ENTRY 0002
$VCBFEND ENTRY 0052
$INITPRT ENTRY 01AA
$INITMOD ENTRY 01AC
$STORAGE ENTRY 0052
STOREMAP ENTRY 0076
$BLOCKCT ENTRY 0052
MAPEND ENTRY 0086
$MEMSIZE ENTRY 0062
$PARTSIZE ENTRY 0066
DATEFMT ENTRY 0064
$MAPAREA ENTRY 0096
$NUMPART ENTRY 01A8
EDXSYS EXTRN
TIMERDDB ENTRY 01AE
TIMER0 ENTRY 01AE
TIMER1 ENTRY 0108
TIMER0IA EXTRN
TIMER1IA EXTRN
DMVJL ENTRY 01F8
DM0DB ENTRY 0226
D49624AT EXTRN
DISKID00 EXTRN
DMIPL ENTRY 03C4
TERMDEFS ENTRY 04CC
FIRSTCCB ENTRY 04CC
WAIT EXTRN
ATTACH EXTRN
KBTASK EXTRN
$SYSLOG ENTRY 052E
IA4979 EXTRN
ID4979 EXTRN
$SYSPRTR ENTRY 0FAC

```

Figure A-2. Assembly statistics and listing (5 of 6)

```

DSPLY1    ENTRY    0708
IA4978    EXTRN
TRASCII   EXTRN
$SYSLOGA  ENTRY    08CC
IATTY     EXTRN
IOTERM    EXTRN
TREBASC   EXTRN
$SYSLOGB  ENTRY    0ADE
IAACCA    EXTRN
IO3101    EXTRN
LINEPRTR  ENTRY    0D94
IA4973    EXTRN
IO4974    EXTRN
IA4974    EXTRN
$SYSCOM   ENTRY    115A
$EDXPTCH  ENTRY    117A

```

COMPLETION CODE = -1

Figure A-2. Assembly statistics and listing (6 of 6)

```

JUMP      ENDJOB,GT,4
PROGRAM   $LINK,EDX002
NOMSG
PARM      $SYSPRTR
DS        LINKCNTL,EDX002
DS        LEWORK1,EDX002
DS        LEWORK2,EDX002
EXEC

```

Figure A-3. Loading link editor

```

$LINK EXECUTION CONTROL RECORDS
FROM LINKCNTL,EDX002
*
* EVENT DRIVEN EXECUTIVE - VERSION 3, MODIFICATION LEVEL 0
*
*****
* COMMENTS MAY BE INCLUDED BY AN '*' IN COLUMN 1 *
* USE THIS TECHNIQUE TO OMIT UNNEEDED MODULES *
*****
OUTPUT    SUPVLINK,EDX002    ENTRY=$START
*
*****
* SUPERVISOR SUPPORT *
*****
*
INCLUDE   EDXSYS,XS3002    *O* SYSTEM TABLES AND WORK AREAS
INCLUDE   ASMOSJ,EDX002    *O* OUTPUT FROM USER SYSTEM GENERATION
INCLUDE   EDXSVCX,XS3002   *O,K* TASK SUPERVISOR (XL)
* INCLUDE EDXSVCXU,XS3002  *O,L* TASK SUPERVISOR (UN-XL)
INCLUDE   $DEBUGNC,XS3002  *G* RESIDENT $DEBUG SUPPORT
INCLUDE   EDXALU,XS3002    *O* EDL INSTRUCTION EMULATOR
INCLUDE   EDXSTART,XS3002  *O* INITIALIZATION & ERROR HANDLER
*
*****
* DEVICE SUPPORT -- DISK(ETTE)S *
*****
*
INCLUDE   DISKIO,XS3002    *M* BASIC DISK(ETTE) SUPPORT
INCLUDE   D49624,XS3002    *M* 4962/4964 DISK(ETTE) SUPPORT
* INCLUDE D4963A,XS3002    *M* 4963 SUBSYSTEM SUPPORT
* INCLUDE   D4966A,XS3002    *M* 4966 MAGAZINE SUPPORT
*
*****
* DEVICE SUPPORT -- TAPE *
*****
*
INCLUDE   D4969A,XS3002    *M* BASIC TAPE SUPPORT
*
*****
* DEVICE SUPPORT -- TERMINALS *
*****
*
INCLUDE   EDXTIO,XS3002    *I,K* BASIC TERMINAL SUPPORT (XL)
* INCLUDE   EDXTIOU,XS3002  *I,L* BASIC TERMINAL SUPPORT (UN-XL)

```

Figure A-4. Link control file (1 of 8)

```

INCLUDE EDXTRMQU,XS3002 *1,K* ENQT/DEQT & TERMINAL QUEUEING (XL)
* INCLUDE EDXTRMQU,XS3002 *1,L* ENQT/DEQT & TERMINAL QUEUEING (UN-XL)
INCLUDE IOS4979,XS3002 *M,K* 4978/4979 DISPLAY SUPPORT (XL)
* INCLUDE IOS4979U,XS3002 *M,L* 4978/4979 DISPLAY SUPPORT (UN-XL)
INCLUDE IOS4974,XS3002 *M,K* 4973/4974 PRINTER SUPPORT (XL)
* INCLUDE IOS4974U,XS3002 *M,L* 4973/4974 PRINTER SUPPORT (UN-XL)
INCLUDE IOSTERM,XS3002 *2* REQUIRED FOR TTY, ACCA, 4013 & 2741
INCLUDE IOSSTY,XS3002 *M* ASR 33/35 TELETYPEWRITER SUPPORT
INCLUDE IOSACCA,XS3002 *3* ASCII ACCA TERMINAL SUPPORT
INCLUDE IOS3101,XS3002 *M,O* 3101 BLOCK MODE SUPPORT
* INCLUDE IOSSIS1,XS3002 *M* SERIES/1 - SERIES/1 SUPPORT
* INCLUDE IOSGP1B,XS3002 *M* GPIB SUPPORT
* INCLUDE IOS4013,XS3002 *M* DIGITAL I/O TERMINAL SUPPORT
* INCLUDE IOS2741,XS3002 *M* 2741 TERMINAL SUPPORT
* INCLUDE IOSVIRT,XS3002 *M,N* VIRTUAL TERMINAL SUPPORT
*
*****
* TERMINAL SUPPORT -- TRANSLATION TABLES *
*****
*
INCLUDE TRASCII,XS3002 *4,P* TELETYPEWRITER TRANSLATION
INCLUDE TREBASC,XS3002 *3,P* MIRROR IMAGE ASCII TRANSLATION
* INCLUDE TREBCD,XS3002 *5* 2741 EBDC TRANSLATION
* INCLUDE TRCRSP,XS3002 *5* 2741 CORRESPONDENCE TRANSLATION
*
*****
* TERMINAL SUPPORT -- SPOOLING *
*****
*
INCLUDE IOSPOOL,XS3002 *M* SPOOLING SUPPORT
*
*****
* DEVICE SUPPORT -- TIMERS *
*****
*
INCLUDE EDXTIMER,XS3002 *6* 4953/4955 TIMER (7840) SUPPORT
* INCLUDE EDXTIMR2,XS3002 *6* 4952 TIMER SUPPORT
*
*****
* DEVICE SUPPORT -- BINARY SYNCHRONOUS COMMUNICATIONS *
*****
*
* INCLUDE BSCAM,XS3002 *7,K* BISYNC COMM. ACCESS SUPPORT (XL)
* INCLUDE BSCAMU,XS3002 *7,L* BISYNC COMM. ACCESS SUPPORT (UN-XL)
* INCLUDE TPCOM,XS3002 *6* HOST COMMUNICATION SUPPORT
*
*****
* DEVICE SUPPORT -- SENSOR INPUT/OUTPUT *
*****
*
* INCLUDE SBCOM,XS3002 *9* BASIC SENSOR I/O SUPPORT
* INCLUDE IOLOADER,XS3002 *9,K* SENSOR I/O DEVICE OPEN (XL)
* INCLUDE IOLOADERU,XS3002 *9,L* SENSOR I/O DEVICE OPEN (UN-XL)
* INCLUDE SBAI,XS3002 *M* ANALOG INPUT SUPPORT
* INCLUDE SBAO,XS3002 *M* ANALOG OUTPUT SUPPORT
* INCLUDE SBDIDG,XS3002 *M* DIGITAL INPUT/OUTPUT SUPPORT
* INCLUDE SBPI,XS3002 *M* PROCESS INTERRUPT SUPPORT
*
*****
* DEVICE SUPPORT -- EXIO CONTROL *
*****
*
* INCLUDE IOSEXIO,XS3002 *M* EXIO DEVICE CONTROL SUPPORT
*
*****
* SYSTEM SUPPORT -- ERROR LOGGING *
*****
*
* INCLUDE SYSLOG,XS3002 *A* I/O ERROR LOGGING
* INCLUDE NOSYSLOG,XS3002 *A* NO I/O ERROR LOGGING
* INCLUDE CIRCBUFF,XS3002 *B* PROGRAM/MACHINE CHECK LOGGING
*
*****
* OPTIONAL FUNCTION SUPPORT *
*****
*
* INCLUDE RLOADER,XS3002 *C,K* RELOCATING PROGRAM LOADER (XL)
* INCLUDE RLOADERU,XS3002 *C,L* RELOCATING PROGRAM LOADER (UN-XL)
* INCLUDE EDXFLOAT,XS3002 *O* FLOATING POINT ARITHMETIC
* INCLUDE NOFLOAT,XS3002 *D* FOR SYSTEMS WITHOUT FLOATING POINT
* INCLUDE EBFLCVT,XS3002 *E* EBCDIC/FLOATING POINT CONV.
* INCLUDE QUEUEIO,XS3002 *F* QUEUE PROCESSING SUPPORT
*
*****
* SYSTEM SUPPORT -- INITIALIZATION *
*****
*
* INCLUDE EDXINIT,XS3002 *H* SUPERVISOR INITIALIZATION
* INCLUDE DISKINIT,XS3002 *M* DISK(ETTE) INITIALIZATION

```

Figure A-4. Link control file (2 of 8)

```

* INCLUDE TAPEINIT,XS3002 *M* TAPE INITIALIZATION
INCLUDE LOADINIT,XS3002 *C* PROGRAM LOADER INITIALIZATION
* INCLUDE RW4963ID,XS3002 *M* 4963 FIXED HEAD REFRESH SUPPORT
INCLUDE TERMINIT,XS3002 *1* TERMINAL INITIALIZATION
INCLUDE INIT4978,XS3002 *M* 4978 DISPLAY INITIALIZATION
* INCLUDE INIT4013,XS3002 *M* DIGITAL I/O TERMINAL INITIALIZATION
INCLUDE $ACCARAM,XS3002 *3* ACCA MULTI-LINE ADAPTER RAM LOAD
* INCLUDE BSCINIT,XS3002 *7* BISYNC (BSCAM) INITIALIZATION
* INCLUDE TPINIT,XS3002 *8* HCF (TPCOM) INITIALIZATION
INCLUDE TIMRINIT,XS3002 *6* 4953/4955 TIMER INITIALIZATION
* INCLUDE CLOKINIT,XS3002 *6* 4952 TIMER INITIALIZATION
* INCLUDE SBIOINIT,XS3002 *M* SENSOR I/O INITIALIZATION
* INCLUDE EXIOINIT,XS3002 *M* EXIO INITIALIZATION
* INCLUDE SIS1INIT,XS3002 *%,Q* SIS1 INITIALIZATION
END

```

\*\*\*\*\* UNRESOLVED EXTERNAL REFERENCES

```

WXTRN EXOPEN
WXTRN SAOA
WXTRN BSCENTRY
WXTRN SDIX
WXTRN SBPI
WXTRN SDIS
WXTRN SDDX
WXTRN SAIX
WXTRN SDOOS
WXTRN SAIS
WXTRN $BSCFDD%
WXTRN SDI
WXTRN SAOX
WXTRN SDOP
WXTRN SDO
WXTRN SAI
WXTRN IOVIRT
WXTRN EXIO
WXTRN STP
WXTRN SAO
WXTRN $TPDVADR
WXTRN SDIA
WXTRN $EXI0DD%
WXTRN SDDA
WXTRN SAIA
WXTRN IOLQAD
WXTRN $PROG1
WXTRN D4963IH1
WXTRN CNTLBUSY
WXTRN VRY4966
WXTRN D4966B
WXTRN TAPEIO
WXTRN VRY4969
WXTRN CNTLEND
WXTRN RD2741
WXTRN RD4013
WXTRN WR2741
WXTRN WR4013
WXTRN EXIOCLEN
WXTRN DEQBSC
WXTRN ACLOSE
WXTRN IOUNLOAD
WXTRN $SVCSIA
WXTRN TPINIT
WXTRN INIT4013
WXTRN BSCINIT
WXTRN TAPEINIT
WXTRN SBIOINIT
WXTRN $SVCLSB
WXTRN EXIOINIT
WXTRN RW4963ID
WXTRN D4963ATN
WXTRN DDBFIX
WXTRN CCBFIX

```

OUTPUT NAME=	SUPVLINK	ADDR	LENGTH
ESD TYPE	LABEL		
CSECT	EDXSYS	0000	052E
ENTRY	\$START	0230	
ENTRY	RETURN	0238	
ENTRY	\$DMDDB	0260	
ENTRY	\$IPLVOL	0262	
ENTRY	\$TIMRTBL	0264	
ENTRY	\$TESTADR	0268	
ENTRY	\$TPDOB	026A	
ENTRY	\$BSCADDR	0290	
ENTRY	EDXFLAGS	0298	
ENTRY	SVCFLAGS	029A	
ENTRY	\$SBPITAB	029C	
ENTRY	LCBA	029E	
ENTRY	\$DMVOL	02A0	
ENTRY	SVCBFIN	02B0	
ENTRY	SVCBFOUT	02B2	

Figure A-4. Link control file (3 of 8)

ENTRY	SVCRTN	02B4	
ENTRY	SVCL1	02B6	
ENTRY	SVCLT	02B6	
ENTRY	SVCL2	02BE	
ENTRY	SVCL3	02C6	
ENTRY	SVCLSB	02CE	
ENTRY	SVC IAR	02CE	
ENTRY	SVCAKR	02D0	
ENTRY	SVCLSR	02D2	
ENTRY	SVCRO	02D4	
ENTRY	SVCRI	02D6	
ENTRY	SVCR2	02D8	
ENTRY	SVCR3	02DA	
ENTRY	SVCR4	02DC	
ENTRY	SVCR5	02DE	
ENTRY	SVCR6	02E0	
ENTRY	SVCR7	02E2	
ENTRY	SVC I IAR	02E4	
ENTRY	SVCILSB	02E4	
ENTRY	SVC IAKR	02E6	
ENTRY	SVCILSR	02E8	
ENTRY	SVC IRO	02EA	
ENTRY	SVCIR1	02EC	
ENTRY	SVCIR2	02EE	
ENTRY	SVCIR3	02F0	
ENTRY	SVCIR4	02F2	
ENTRY	SVCIR5	02F4	
ENTRY	SVCIR6	02F6	
ENTRY	SVCIR7	02F8	
ENTRY	UNCHAKR1	02FA	
ENTRY	UNCHSAV6	02FC	
ENTRY	SVC P ARMS	0300	
ENTRY	CMDTABLE	0306	
CSECT	\$EDXDEF	052E	122A
ENTRY	SVCBF	0530	
ENTRY	\$SVCIBUF	0530	
ENTRY	SVCBFEND	0580	
ENTRY	\$STORAGE	0580	
ENTRY	\$BLOCKCT	0580	
ENTRY	\$MEMSIZE	0590	
ENTRY	DATEFMT	0592	
ENTRY	\$PARTSZE	0594	
ENTRY	STOREMAP	05A4	
ENTRY	MAPEND	05B4	
ENTRY	\$MAPAREA	05C4	
ENTRY	\$NUMPART	0686	
ENTRY	\$INITPR	0688	
ENTRY	\$INITMOD	068A	
ENTRY	TIMERDDB	068C	
ENTRY	TIMERO	068C	
ENTRY	TIMER1	0686	
ENTRY	DMVOL	06D6	
ENTRY	DMDDB	0704	
ENTRY	DMIPL	08A2	
ENTRY	FIRSTCCB	09AA	
ENTRY	TERMDEFS	09AA	
ENTRY	\$SYSLJG	0A0C	
ENTRY	DISPLAY1	0BE6	
ENTRY	\$SYSLOGA	0DAA	
ENTRY	\$SYSLOGB	0FBC	
ENTRY	\$SYSPRTR	1272	
ENTRY	MATRIX	148A	
ENTRY	\$SYSCOM	1638	
ENTRY	\$EDXPTCH	1658	
CSECT	EDXSVCX	1758	0880
ENTRY	SVC	1758	
ENTRY	SVCA	178A	
ENTRY	SETBUSY	17D4	
ENTRY	SVC I	17DC	
ENTRY	WAIT	1828	
ENTRY	ENQ	1936	
ENTRY	DEQ	1888	
ENTRY	PCST	1898	
ENTRY	ATTACHX	19DE	
ENTRY	ATTACH	19E2	
ENTRY	DETACH	1A44	
ENTRY	SUPEXIT	1A76	
ENTRY	SUPEXTRL	1ABC	
ENTRY	SUPLVLXO	1AE4	
ENTRY	SATTACH	1B96	
ENTRY	SDETACH	1RE0	
ENTRY	SCWAIT	1C02	
ENTRY	SWAIT	1C16	
ENTRY	SPOST	1C2E	
ENTRY	SRESETEV	1C4E	
ENTRY	SENQ	1C5A	
ENTRY	SDEJ	1CA0	
ENTRY	STPTASK1	1CC0	
ENTRY	STPTASK2	1CC4	

Figure A-4. Link control file (4 of 8)

ENTRY	UNCHAIN	1CE3	
ENTRY	\$CP	1D40	
ENTRY	LPGMXP1	1DDB	
ENTRY	LPGMXP2	1EC4	
CSECT	\$OBUGNUC	2008	018A
ENTRY	\$TESTCOM	2008	
ENTRY	STESTIN	201E	
ENTRY	STESTOUT	2086	
ENTRY	\$TRCSIA	210A	
ENTRY	\$TRCLSB	217A	
CSECT	EDXALU	2192	085E
ENTRY	#IFB	2192	
ENTRY	#IFDW	2196	
ENTRY	#IFW	219A	
ENTRY	#IFTTEST	219C	
ENTRY	\$COMPE	21CA	
ENTRY	\$COMPNE	21DE	
ENTRY	\$FINDE	21F0	
ENTRY	\$FINDNE	2216	
ENTRY	CGOTO	2234	
ENTRY	BRANCH	2250	
ENTRY	\$EXEC	2266	
ENTRY	#NOP	226A	
ENTRY	CMOSETUP	226C	
ENTRY	CMO\$TEST	2272	
ENTRY	SDOLOOP	22CA	
ENTRY	SCONTINU	22DA	
ENTRY	SAV222CR	22E8	
ENTRY	SAV424CR	22F0	
ENTRY	SAV444CR	2304	
ENTRY	SAV224CR	230E	
ENTRY	SAX222	232E	
ENTRY	SA222C	2334	
ENTRY	SA222	2338	
ENTRY	SSX222	234A	
ENTRY	SS222C	234E	
ENTRY	SS222	2352	
ENTRY	SM222	2364	
ENTRY	SM222C	2368	
ENTRY	SD222	237C	
ENTRY	SD222C	2380	
ENTRY	GETPAR3	23C6	
ENTRY	GETCNT	23DE	
ENTRY	SA424	23EC	
ENTRY	SA424C	23F6	
ENTRY	SS424	241A	
ENTRY	SS424C	241E	
ENTRY	SM424	2442	
ENTRY	SM424C	2446	
ENTRY	SD424	2460	
ENTRY	SD424C	2464	
ENTRY	SX444	248E	
ENTRY	SX444C	2494	
ENTRY	SX224R	24B2	
ENTRY	SX224CR	24BA	
ENTRY	SD422R	24F2	
ENTRY	SD422CR	24F8	
ENTRY	MOV1	2654	
ENTRY	MOV1C	265C	
ENTRY	AND1	267A	
ENTRY	AND1XX	2684	
ENTRY	IOR1	2690	
ENTRY	IOR1XX	2694	
ENTRY	EOR1	269C	
ENTRY	EOR1XX	26A4	
ENTRY	SHR1	26AC	
ENTRY	SHR1XX	26C0	
ENTRY	SHL1	26D2	
ENTRY	SHL1XX	26E2	
ENTRY	MOV2	26FC	
ENTRY	MOV2C	2704	
ENTRY	AND2	2726	
ENTRY	AND2XX	2730	
ENTRY	IOR2	273C	
ENTRY	IOR2XX	2740	
ENTRY	EOR2	2748	
ENTRY	EOR2XX	2750	
ENTRY	SHR2	2758	
ENTRY	SHR2XX	2768	
ENTRY	SHL2	2776	
ENTRY	SHL2XX	2786	
ENTRY	MOV4	27A0	
ENTRY	MOV4C	27B8	
ENTRY	AND4	27E2	
ENTRY	AND4XX	27EE	
ENTRY	IOR4	2804	
ENTRY	IOR4XX	2808	
ENTRY	EOR4	2810	

Figure A-4. Link control file (5 of 8)

ENTRY	EOR4XX	2818	
ENTRY	SHR4	2820	
ENTRY	SHR4XX	2830	
ENTRY	SHL4	283C	
ENTRY	SHL4XX	284C	
ENTRY	MOVEXP	28DE	
ENTRY	SCALL	2942	
ENTRY	SRETURN	2966	
ENTRY	BFRCK	298A	
ENTRY	USER	29BC	
ENTRY	EDXLOJP	29E0	
CSECT	EDXSTART	29F0	05C6
ENTRY	INITTASK	29F0	
ENTRY	PCHKSLA	2C24	
ENTRY	PCHKLSB	2E1A	
ENTRY	SFTKSLA	2E3E	
ENTRY	STARTPGM	2F88	
CSECT	DISKID	2FB6	06D8
ENTRY	DISKRW	2FB6	
ENTRY	DISKRW05	3072	
ENTRY	TAPE060	30C4	
ENTRY	DISKID00	313E	
ENTRY	DCBRETRN	3224	
ENTRY	DISKFLIH	32EA	
ENTRY	DFLIH04	331E	
ENTRY	DISKERR1	332A	
ENTRY	DISKERR2	332E	
ENTRY	DISKERR3	3332	
ENTRY	DISKERR5	3336	
ENTRY	DISKERR6B	333A	
ENTRY	DISKERR7	333E	
ENTRY	DISKER13	3366	
ENTRY	DISKPOST	3384	
ENTRY	VARYON	3394	
ENTRY	VARYOFF	339C	
ENTRY	VARYWORD	34F2	
ENTRY	VARYQCB	34F4	
ENTRY	VARYDSCB	34FE	
CSECT	D49624	368E	05B2
ENTRY	D49624AT	368E	
ENTRY	D4962IH1	37D4	
ENTRY	DFLIH50	39B2	
ENTRY	DFLIH54	39FA	
ENTRY	DISKAITN	3A24	
ENTRY	DATTN00	3A2E	
CSECT	EDXTIO	3C40	1348
ENTRY	SDPEND	3E52	
ENTRY	PRSKSP	3E80	
ENTRY	CURCTL	3F2E	
ENTRY	CTLXFER	3FE6	
ENTRY	PRTEXT	404E	
ENTRY	NXTCOMD	40D0	
ENTRY	RDTXTL	40EE	
ENTRY	RDTXT	40F2	
ENTRY	QUESTION	41EA	
ENTRY	PRTNUM2S	4246	
ENTRY	PRTNUM2	424C	
ENTRY	PRTNUM4S	426C	
ENTRY	PRTNUM4	4272	
ENTRY	GETVAL2	4348	
ENTRY	GETVAL4	4366	
ENTRY	KBTASK	4456	
ENTRY	ENDATTN	45D2	
ENTRY	TERMOUT	4718	
ENTRY	TERMINT	48F0	
ENTRY	DECSCAN	480E	
ENTRY	FLDCLEAR	4C1C	
ENTRY	BDCWORD	4C38	
ENTRY	DCBWORD	4DCA	
ENTRY	EBBICVT	4E6C	
CSECT	EDXTERMQ	4F88	03A2
ENTRY	ENQT	4F88	
ENTRY	DEQT	511C	
ENTRY	QUTERMIN	5172	
ENTRY	QUTERM	517A	
ENTRY	DQTERM	51EC	
ENTRY	DQTERMIN	5224	
ENTRY	DQTERMB	525E	
ENTRY	DEQTERMS	52CE	
CSECT	IOS4979	532A	090A
ENTRY	IO4979	532A	
ENTRY	IO4978	532A	
ENTRY	IA4979	5B20	
ENTRY	IA4978	5B20	
CSECT	IOS4974	5C34	02E0
ENTRY	IO4974	5C34	
ENTRY	IO4973	5C34	
ENTRY	IA4973	5ED6	
ENTRY	IA4974	5ED6	

Figure A-4. Link control file (6 of 8)

CSECT	IOSTERM	5F14	021A
ENTRY	IOTERM	5F14	
CSECT	IOSTTY	612E	02CE
ENTRY	WRTTY	612E	
ENTRY	RDTTY	6168	
ENTRY	IATTY	61CA	
CSECT	IOSACCA	63FC	0596
ENTRY	WRACCA	63FC	
ENTRY	RDACCA	654A	
ENTRY	IAACCA	6702	
ENTRY	IADELAY	6878	
ENTRY	ACCALS	6882	
CSECT	IOS3101	6992	0788
ENTRY	IO3101	6992	
CSECT	ASCIIITAB	714A	0202
ENTRY	TRASCII	714A	
CSECT	EBASCII	734C	0202
ENTRY	TREBASC	734C	
CSECT	IOSPOOL	754E	04DA
ENTRY	IOSPNQT	754E	
ENTRY	IOSPDQT	7676	
ENTRY	IOSPWR	774C	
ENTRY	IOSPCMD	78DC	
ENTRY	IOSPCLOS	7932	
ENTRY	IOSPEND	795A	
ENTRY	IOSPTBL	79EE	
CSECT	EDXTIMER	7A28	0484
ENTRY	TIMER0IA	7A28	
ENTRY	TIMER1IA	7A98	
ENTRY	SETIMER	7B74	
ENTRY	WAITIMER	7C6A	
ENTRY	INTIMEX	7C8E	
ENTRY	INTIME	7C9C	
ENTRY	GTIMDATE	7CF4	
ENTRY	PRINTIME	7D20	
ENTRY	WHATIME	7D9E	
ENTRY	SETCLOCK	7DC8	
CSECT	SYSLOG	7EAC	0162
ENTRY	LCB	7EAC	
ENTRY	\$LOGIA	7EBC	
ENTRY	\$LOGTSK	7F2C	
ENTRY	\$SLOGIA	7FC6	
ENTRY	\$SLOGTSK	7FD2	
ENTRY	\$SLOGPRM	8000	
CSECT	CIRCBUFF	800E	00FC
ENTRY	CIRSTR	800E	
ENTRY	CIRIN	8010	
ENTRY	CIREND	8012	
ENTRY	CIRCNT	8014	
ENTRY	CIRESIZ	8016	
ENTRY	CIRESTR	8018	
CSECT	RLOADER	810A	127E
ENTRY	LOADPGM	810A	
ENTRY	LPGMXP	8150	
ENTRY	LOADPGMO	81BC	
ENTRY	LOADEXIT	8386	
ENTRY	LPGMXPB	83FA	
ENTRY	ENDCODE	84AA	
ENTRY	LOADQCB	84AC	
ENTRY	LOADORG	84B6	
ENTRY	LCMDKEY	84DA	
ENTRY	LCMDTGT	84DC	
ENTRY	LOADFHFL	84E2	
ENTRY	GETMAIN	88F0	
ENTRY	FREEMAIN	89DA	
ENTRY	\$ACTIVE	8A7E	
ENTRY	GOTOTABL	8D50	
ENTRY	\$CANCEL	8DCC	
ENTRY	STOP	8F96	
ENTRY	STOPTASK	90DA	
CSECT	EDXFLOAT	9388	0264
ENTRY	FADD010	9388	
ENTRY	FADD000	9388	
ENTRY	FADD100	9388	
ENTRY	FADD001	9396	
ENTRY	FADD011	93A4	
ENTRY	FADD101	93B2	
ENTRY	FADD110	93C0	
ENTRY	FADD111	93CE	
ENTRY	FSUB000	93DC	
ENTRY	FSUB100	93DC	
ENTRY	FSUB001	93EA	
ENTRY	FSUB010	93F8	
ENTRY	FSUB011	9406	
ENTRY	FSUB101	9414	
ENTRY	FSUB110	9422	
ENTRY	FSUB111	9430	
ENTRY	FLOATERR	9468	
ENTRY	FMPY000	9478	

Figure A-4. Link control file (7 of 8)

```

ENTRY  FMPY001  9486
ENTRY  FMPY010  9494
ENTRY  FMPY011  94A2
ENTRY  FMPY110  94B0
ENTRY  FMPY100  94B0
ENTRY  FMPY111  94BE
ENTRY  FMPY101  94BE
ENTRY  FDIV000  94CC
ENTRY  FDIV001  94DA
ENTRY  FDIV010  94E8
ENTRY  FDIV011  94F6
ENTRY  FDIV110  9504
ENTRY  FDIV100  9504
ENTRY  FDIV111  9512
ENTRY  FDIV101  9512
ENTRY  FLTCONV  9520
ENTRY  MOVFP4   9574
ENTRY  MOVFP8   958A
ENTRY  IFFLOAT  95B8
ENTRY  IFFLOATL 95D0
ENTRY  EDXFLEND 95E8
CSECT  EBFLCVT  95EC      065C
ENTRY  EBFLD3L  95EC
ENTRY  EBFLSTD  95EC
ENTRY  FLEBDBL  9908
ENTRY  FLEBSTD  9908
CSECT  QIO      9C48      0104
CSECT  EDXINIT  9D4C      018C
ENTRY  START    9D4C
ENTRY  $EDXINIT 9D76
ENTRY  INITEXIT 9DDA
ENTRY  $SBIDINT 9E76
ENTRY  $TERMINT 9E78
ENTRY  $DISKINT 9E7C
ENTRY  $TAPEINT 9E7E
ENTRY  $4978INT 9E80
ENTRY  $4013INT 9E82
ENTRY  $LOADINT 9E84
ENTRY  $PGMCINT 9E86
ENTRY  $HDSTINT 9E8A
ENTRY  $BSCAINT 9E8C
ENTRY  $EXIOINT 9E8E
ENTRY  $TIMEINT 9E90
ENTRY  INITFEAT 9F04
CSECT  DISKINIT 9F08      0098
ENTRY  DSKPREP2 A136
ENTRY  DSKINIT1 A23E
ENTRY  D66INIT  A344

ENTRY  PREPIDCB A406
ENTRY  DISKBUFR A418
ENTRY  DSOPEN   A69A
ENTRY  GETVOL   A74E
ENTRY  ENQVOL   AB60
ENTRY  DEQVOL   ABC6
ENTRY  $DSNFND  AC02
ENTRY  $DSBIODA AC04
ENTRY  $DSBVOL  AC06
ENTRY  $DSBLIB  AC08
ENTRY  $DSIOERR AC0A
ENTRY  $DSNVTOC AC0C
ENTRY  $$EXIT   AC0E
ENTRY  $DSDCEA AC10
ENTRY  DS$      AC5C
CSECT  LOADINIT ACA0      0610
CSECT  TERMINIT B2B0      02D0
ENTRY  NEXTERM  B412
ENTRY  TERMERRX B478
CSECT  INIT4978 B580      06A0
CSECT  $ACCARAM BC20      01F6
CSECT  TIMRINIT BE16      0158
MODULE TEXT LENGTH= BF6E, RLD COUNT= 3131
SUPVLINK ADDED TO EDX002

```

```

$LINK COMPLETION CODE= -1
AT ON

```

```

$LINK ENDED AT
JUMP ENDJOB,GT,4

```

Figure A-4. Link control file (8 of 8)

```
PROGRAM $UPDATE,EDX002
NOMSG
PARM $SYSRTR SUPVLINK,EDX002 $EDXNUCN,EDX002 YES
EXEC
$EDXNUCN STORED

$UPDATE ENDED AT
LABEL ENDJOB
```

**Figure A-5. End of SYSGEN**

## Appendix B. Program Preparation Listings

### EDX ASSEMBLER STATISTICS

SOURCE INPUT - STATSRC,EDX002  
 WORK DATA SET - ASMWORK,EDX002  
 OBJECT MODULE - ASMOUT,EDX002  
 STATEMENTS PROCESSED - 70

NO STATEMENTS FLAGGED

LOC	+0	+2	+4	+6	+8	SOURCE STATEMENT	STATSRC ,EDX002	(5719-XX4)-V3.0.0	
0000	0008	07D9	06C7	09C1	0440	XMPLSTAT	PROGRAM	START	00000010
000A	0000	05E4	0370	0000	0000				
0014	06E6	0000	0000	0000	0100				
001E	06E4	0000	0000	0000	0664				
0028	0000	0000	0000	0000	0000				
0032	0000								
0034	0000	0300	0000	0000	0000	IMAGEBUF	EXTRN	\$IMOPEN,\$IMDEFN,\$IMPROT,\$IMDATA	00000020
003E	0000	0000	0000	0000	0000		BUFFER	769,BYTES	00000030
0036	0000								
0038	0E0D	E5C9	C4C5	D6F1	68C1	DSETNAME	TEXT	'VIDEO1,ASMVOL'	00000040
0042	E2D4	E5D6	D340						
0048	4040	4040	4040	4040	8000	IUCB1	IOCB	NHIST=0	00000050
0052	00FF	0000	7FFF	0000	0000				
005C	4040	4040	4040	4040	8800	IUCB2	IOCB	SCREEN=STATIC	00000060
0066	00FF	0000	7FFF	0000	0000				
0070	0002	0403	C5D5	C440	05B2	ATTNLIST		(END,OUT,\$PF,STATIC)	00000070
007A	0403	58D7	C640	05BA					
0082	9025	0348				START	ENQT	IUCB1	00000130
0086	B02A	0001	000F	8026	1414		PRINTTEXT	'CLASS ROSTER PROGRAM',SPACES=15,LINE=1	00000140
0090	C3D3	C1E2	E240	D9D6	F2E3				
009A	C5D9	4007	D9D6	C709	C1D4				
00A4	902A	0002	0000	8026	2020		PRINTTEXT	'HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2	00000150
00AE	C8C9	E340	7DC1	E3E3	D57D				
00B8	40C1	D5C4	40C5	D5E3	C5D9				
00C2	407D	C5D5	C47D	E3D6	40C5				
00CC	D5C4								
00CE	8026	0C08	E3C8	C540	D7D9		PRINTTEXT	'THE PROGRAM'	00000160
00D8	D6C7	D9C1	D440						
00DE	902A	0007	0000	8026	201F		PRINTTEXT	'HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2	00000170
00E8	C8C9	E340	C1D5	E840	D7D9				
00F2	D6C7	D9C1	D440	C6E4	D5C3				
00FC	E3C9	D6D5	40D2	C5E8	40E3				
0006	D640								
0408	8026	1A1A	40C2	D9C9	D5C7		PRINTTEXT	' BRING UP THE ENTRY SCREEN'	00000180
0412	40E4	D740	E3C8	C540	C5D5				
041C	E3D9	E940	E2C3	D9C5	C5D5				
0426	00B2								
0428	0018	05C2				CHECK	DEQT		00000190
042C	A0A2	05C2	0001	055C			WAIT	ATTNECB,RESET	00000240
0434	C29E	0000	033A	0038		GETIMAGE	IF	00000250	
043C	A0A2	05E6	FFFF	0472			CALL	\$IMOPEN,(DSETNAME),(IMAGEBUF)	00000260
0444	005C	05CA	05E6				IF	(XMPLSTAT+2,NE,-1)	00000270
044A	8026	1A1A	7CC9	D4C1	C7C5		MOVE	ERRCODE,XMPLSTAT+2	00000280
0454	40D6	D7C5	D540	C5D9	D9D6		PRINTTEXT	'@IMAGE OPEN ERROR, CODE = '	00000290
045E	D968	40C3	D6C4	C540	7E40				
0468	0028	05CA	0001				PRINTNUM	ERRCODE	00000300
046E	00A0	05CC					GOTO	ERRQUERY	00000310
0472							ENDIF		00000320
0472	C29E	0000	0348	0038			CALL	\$IMDEFN,(IOCB1),(IMAGEBUF)	00000330
047A	9025	035C					ENQT	IOCB2	00000340
047E	1430						TERMCTRL	BLANK	00000350
0480	C29E	0000	0038	0000			CALL	\$IMPROT,(IMAGEBUF),0	00000360
0488	B19E	0000	0038				CALL	\$IMDATA,(IMAGEBUF)	00000370
048E	B02A	0004	000B				PRINTTEXT	LINE=4,SPACES=11	00000380
0494	1C30						TERMCTRL	DISPLAY	00000390
0496	2030					WAIT TONE	WAIT	KEY	00000400
0498	00A1	05E6	0004	050E	04A8		GOTO	(READ,E1,E2,E3,E4),XMPLSTAT+2	00000410
04A2	0432	043C	04C6						
04A8	805C	05C8	0006			E1	MOVE	LINE#R,6	00000420
04AE	00A0	04CC					GOTO	DELETE	00000430
0432	805C	05C8	0003			E2	MOVE	LINE#R,11	00000440

Figure B-1. Assembler statistics and listing (1 of 2)

```

0488 00A0 04CC          GOTO      DELETE          00000450
049C 805C 05C8 0010    E3        MOVE      LINENBR,16     00000460
04C2 00A0 04CC          GOTO      DELETE          00000470
04C6 805C 05C8 0015    E4        MOVE      LINENBR,21     00000480
04CC E02A 05C8 0000 F030 0004 DELETE     ERASE     MODE=LINE,TYPE=DATA,LIN=LINENBR 00000490
04D6 2000
04D8 8032 05C8 0001          ADD      LINENBR,1       00000500
04DE E02A 05C8 0000 F030 0004 EFASE     MODE=LINE,TYPE=DATA,LIN=LINENBR 00000510
04EB 2000
04EA 8032 05C8 0001          ADD      LINENBR,1       00000520
04FO E02A 05C8 0000 F030 0004 ERASE     MODE=LINE,TYPE=DATA,LIN=LINENBR 00000530
04FA 2000
04FC 8035 05C8 0002          SUBTRACT LINENBR,2       00000540
0502 A02A 05C8 0003          PRINTEXT LINE=LINENBR,SPACES=5 00000550
0509 1C30          TERMCTRL DISPLAY          00000560
050A 00A0 0496          GOTO     WAITONE         00000570
050F F02A 0002 0037 C026 100F READ      QUESTION   *MORE ENTRIES ? *,LINE=2,SPACES=55,NO=CLEANUP 00000580
0519 D406 D9C5 40C5 D5E3 D9C9
0522 C5E2 406F 4040 802E 0550
052C F02A 0002 0037 F030 0004          ERASE     MODE=LINE,LIN=2,SPACES=55,TYPE=DATA 00000590
0536 2000
0538 F02A 0006 0000 F030 0000          ERASE     MODE=SCREEN,LIN=6       00000600
0542 2000
0544 E02A 0006 0005          PRINTEXT  LIN=6,SPACES=5     00000610
054A 1C30          TERMCTRL DISPLAY          00000620
054C 00A0 0496          GOTO     WAITONE         00000630
0550 F030 0001 2000          CLEANUP   ERASE     MODE=SCREEN,TYPE=ALL     00000640
0556 0032          DEQT
0558 00A0 0382          GOTO     START           00000650
055C 0022 FFFF          ENDIT     PROGSTOP        00000660
0560 5050          DATA    X*5050*          00000670
0562 6060 6060 6060 6060 6060 DASHES    DATA    80C*-*          00000680
0564 0019 05C2 0001          OUT      POST      ATTNECB,1       00000690
0566 0010          ENDATTN  POST      ATTNECB,-1      00000700
0568 0019 05C2 FFFF          STATIC   ENDATTN  POST      ATTNECB,-1      00000710
056A 0010          ECB
056C FFFF 0000 0000          ATTNECB  LINENBR  DATA    F*0*          00000720
056E 0000          ERRCODE  DATA    F*0*          00000730
0570 0000          ERRQUERY QUESTION  *RETRY OPEN ? *,YES=GETIMAGE,NO=ENDIT 00000740
0572 C026 0E0E 7C09 C5E3 D9E8
0574 40D6 D7C5 D540 6F40 C02E
0576 0434 055C
0578 0000 0000 0000 0234 0000          ENDPROG          00000780
057A 0000 0000 0382 05E4 0000
057C 0002 0096 0000 0000 FFFF
057E 0000 0000 0610 0000 0000
0580 0612 E7D4 D7D3 E2E3 C1E3
0582 0000 0000 0000 0000 0000
0584 0000 0000 FFFF 0000 0000
0586 0000 0000 0000 05E4 0000
0588 0000 0000 0000 0000 0000
058A 0000 0000 0000 0000 0000
058C 0000 0000 0000 0000 0000
058E 0000 0000 0000 0000 0000
0590 0000 0000 0234 0000 0000
0592 0000 0000 0664 0000 0000
0594 0000 0000 0000 0000 0001
0596 000A 0000 0000 FFFF 0000
0598 0000 0690 0000 0000 0692
059A 58C1 E3E3 C1E2 D240 0000
059C 0000 0000 0000 0000 0000
059E 0000 FFFF 0000 0000 0000
0600 0000 0000 05E4 0000 0000
0602 0000 0000 0000 0000 0000
0604 0000 0000 0000 0000 0000
0606 0000 C664 0080 0000 0000
0608 0000 0000 0000 0000 0000
060A 0000 0000 0000 0000
060C 0000 0000 0000 0000
060E 0000 0000 0000 0000
0610 0000 0000 0000 0000
0612 0000 0000 0000 0000
0614 0000 0000 0000 0000
0616 0000 0000 0000 0000
0618 0000 0000 0000 0000
061A 0000 0000 0000 0000
061C 0000 0000 0000 0000
061E 0000 0000 0000 0000
0620 0000 0000 0000 0000
0622 0000 0000 0000 0000
0624 0000 0000 0000 0000
0626 0000 0000 0000 0000
0628 0000 0000 0000 0000
062A 0000 0000 0000 0000
062C 0000 0000 0000 0000
062E 0000 0000 0000 0000
0630 0000 0000 0000 0000
0632 0000 0000 0000 0000
0634 0000 0000 0000 0000
0636 0000 0000 0000 0000
0638 0000 0000 0000 0000
063A 0000 0000 0000 0000
063C 0000 0000 0000 0000
063E 0000 0000 0000 0000
0640 0000 0000 0000 0000
0642 0000 0000 0000 0000
0644 0000 0000 0000 0000
0646 0000 0000 0000 0000
0648 0000 0000 0000 0000
064A 0000 0000 0000 0000
064C 0000 0000 0000 0000
064E 0000 0000 0000 0000
0648 0000 0000 0000 0000
0650 0000 0000 0000 0000
0652 0000 0000 0000 0000
0654 0000 0000 0000 0000
0656 0000 0000 0000 0000
0658 0000 0000 0000 0000
065A 0000 0000 0000 0000
065C 0000 0000 0000 0000
065E 0000 0000 0000 0000
0660 0000 0000 0000 0000
0662 0000 0000 0000 0000
0664 0000 0000 0000 0000
0666 0000 0000 0000 0000
0668 0000 0000 0000 0000
066A 0000 0000 0000 0000
066C 0000 0000 0000 0000
066E 0000 0000 0000 0000
0670 0000 0000 0000 0000
0672 0000 0000 0000 0000
0674 0000 0000 0000 0000
0676 0000 0000 0000 0000
0678 0000 0000 0000 0000
067A 0000 0000 0000 0000
067C 0000 0000 0000 0000
067E 0000 0000 0000 0000
0680 0000 0000 0000 0000
0682 0000 0000 0000 0000
0684 0000 0000 0000 0000
0686 0000 0000 0000 0000
0688 0000 0000 0000 0000
068A 0000 0000 0000 0000
068C 0000 0000 0000 0000
068E 0000 0000 0000 0000
0690 0000 0000 0000 0000
0692 0000 0000 0000 0000
0694 0000 0000 0000 0000
0696 0000 0000 0000 0000
0698 0000 0000 0000 0000
069A 0000 0000 0000 0000
069C 0000 0000 0000 0000
069E 0000 0000 0000 0000
06A0 0000 0000 0000 0000
06A2 0000 0000 0000 0000
06A4 0000 0000 0000 0000
06A6 0000 0000 0000 0000
06A8 0000 0000 0000 0000
06AA 0000 0000 0000 0000
06AC 0000 0000 0000 0000
06AE 0000 0000 0000 0000
06B0 0000 0000 0000 0000
06B2 0000 0000 0000 0000
06B4 0000 0000 0000 0000
06B6 0000 0000 0000 0000
06B8 0000 0000 0000 0000
06BA 0000 0000 0000 0000
06BC 0000 0000 0000 0000
06BE 0000 0000 0000 0000
06C0 0000 0000 0000 0000
06C2 0000 0000 0000 0000
06C4 0000 0000 0000 0000
06C6 0000 0000 0000 0000
06C8 0000 0000 0000 0000
06CA 0000 0000 0000 0000
06CC 0000 0000 0000 0000
06CE 0000 0000 0000 0000
06D0 0000 0000 0000 0000
06D2 0000 0000 0000 0000
06D4 0000 0000 0000 0000
06D6 0000 0000 0000 0000
06D8 0000 0000 0000 0000
06DA 0000 0000 0000 0000
06DC 0000 0000 0000 0000
06DE 0000 0000 0000 0000
06E0 0000 0000 0000 0000
06E2 0000 0000 0000 0000
06E4 0000 0000 0000 0000
06E6 0000 0000 0000 0000
06E8 0000 0000 0000 0000
06EA 0000 0000 0000 0000
06EC 0000 0000 0000 0000
06EE 0000 0000 0000 0000
06F0 0000 0000 0000 0000
06F2 0000 0000 0000 0000
06F4 0000 0000 0000 0000
06FA 0000 0000 0000 0000

```

```

END
EXTERNAL/UNDEFINED SYMBOLS

```

```

SVC      WXTRN
SUPEXIT  WXTRN
SETBUSY  WXTRN
$IMOPEN  EXTRN
$IMDEFN  EXTRN
$IMPROT  EXTRN
$IMDATA  EXTRN

```

COMPLETION CODE = -1

Figure B-1. Assembler statistics and listing (2 of 2)

```

$LINK EXECUTION CONTROL RECORDS
FROM LINKSTAT,EDX002
* THIS LINK CONTROL DATA SET SPECIFIES:
* 1.) THE LINKED OUTPUT OBJECT MODULE WILL
* BE STORED IN 'LINKOUT' ON EDX002
* 2.) THE AUTOCALL DATA SET IS '$AUTO' ON
* ASMLIB (SYSTEM SUPPLIED)
* 3.) 'ASMOUT' ON EDX002 IS THE ONLY INPUT
* OBJECT MODULE TO BE INCLUDED
OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
INCLUDE ASMOUT
INCLUDE $IMOPEN,ASMLIB VIA AUTOCALL
INCLUDE $IMGEN,ASMLIB VIA AUTOCALL
INCLUDE $IMDTYPE,ASMLIB VIA AUTOCALL
INCLUDE $$RETURN,ASMLIB VIA AUTOCALL
INCLUDE $UNPACK,ASMLIB VIA AUTOCALL
END
**** UNRESOLVED EXTERNAL REFERENCES
WXTRN SVC
WXTRN SUPEXIT
WXTRN SETBUSY
OUTPUT NAME= LINKOUT
ESD TYPE LABEL ADDR LENGTH
CSECT 0000 06FA
CSECT 06FA 09E8
ENTRY $IMOPEN 06FC
ENTRY $FILE 0908
ENTRY DISKBUFR 095C
ENTRY DSOOPEN 0A7A
ENTRY $DSNFND 0FE2
ENTRY $DSBIDDA 0FE4
ENTRY $DSBVOL 0FE6
ENTRY $DSBLIB 0FE8
ENTRY $DSIOERR 0FEA
ENTRY $DSNVTOC 0FEC
ENTRY $$EXIT 0FEE
CSECT 10E2 0E2A
ENTRY $IMDEFN 10E4
ENTRY $IMPROT 1180
ENTRY $IMDATA 1908
ENTRY $ADDRTBL 1EA8
ENTRY $ATTRTBL 1EF8
CSECT 1FOC 0074
ENTRY $IMDTYPE 1FOE
CSECT 1F80 0028
ENTRY RETURN 1F80
CSECT 1FA8 0040
ENTRY $UNPACK 1FAA
MODULE TEXT LENGTH= 1FE8, RLD COUNT= 1015
LINKOUT ADDED TO EDX002

```

```

$LINK COMPLETION CODE= -1
AT 06:38:12 ON 00/00/00

```

```

$LINK ENDED AT 06:38:12

```

Figure B-2. Link edit listing

```

00010 08 *** TOO MANY POSITIONAL OPERANDS WERE SPECIFIED
00020 08 *** AN INVALID KEYWORD PARAMETER WAS SPECIFIED
00030 08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED
00040 08 *** INVALID NO. OF ELEMENTS IN OPERAND - SHOULD BE 1 OR 2
00050 08 *** INVALID INDEX REGISTER SPECIFICATION - NOT #1 OR #2
00060 08 *** RESULT= OPERAND MUST BE SPECIFIED
00070 08 *** INVALID PRECISION FOR REGISTER OPERATION
00080 08 *** OPERAND 1 IS MISSING
00090 08 *** OPERAND 2 IS MISSING
00100 08 *** 'COUNT' IS NOT ALLOWED WITH INDEX REGISTERS
00110 08 *** INVALID OR UNDEFINED OPERATION CODE
00120 08 *** TASK NAME NOT SPECIFIED
00130 08 *** TOO MANY DATA SETS SPECIFIED
00140 08 *** TOO MANY OVERLAY PROGRAMS SPECIFIED
00150 08 *** INVALID PARAMETER COUNT
00160 08 *** START= OPERAND MUST BE SPECIFIED
00170 08 *** DS#= OPERAND MUST BE SPECIFIED
00180 08 *** DSNAME= OPERAND MUST BE SPECIFIED.
00190 08 *** DSLEN= OPERAND IS INVALID
00200 08 *** INVALID PRIORITY SPECIFICATION
00210 08 *** INVALID LEVEL SPECIFICATION
00220 08 *** OPERAND FIELD IS TOO LARGE
00230 08 *** INVALID PREC= SPECIFICATION
00240 08 *** UNBALANCED PARENTHESIS IN OPERAND
00250 08 *** SYMBOL IS MULTIPLY DEFINED

```

Figure B-3. \$EDXL listing (1 of 4)

```

00260 08 *** SYMBOL EXCEEDS 8 CHARACTERS IN LENGTH
00270 08 *** INVALID SELF-DEFINING TERM
00280 08 *** I/O BUFFER ADDRESS NOT SPECIFIED
00290 08 *** QUERY MESSAGE MUST BE SPECIFIED
00300 08 *** INVALID DS= SPECIFICATION
00310 08 *** INVALID PGM= SPECIFICATION
00320 08 *** INVALID PARM= SPECIFICATION
00330 08 *** INVALID LENGTH= SPECIFICATION
00340 08 *** TEXT MESSAGE IS NOT A VALID CHARACTER STRING
00350 08 *** INVALID SYNTAX IN OPERAND FIELD
00360 08 *** NULL OR INVALID BRANCH TABLE ENTRY
00370 08 *** EVENT NAME NOT SPECIFIED
00380 08 *** COPY CODE MODULE NOT DEFINED
00390 08 *** A COPY STATEMENT IS NOT ALLOWED WITHIN COPY CODE
00400 08 *** EITHER YES= OR NO= MUST BE CODED
00410 08 *** INVALID PROMPT= SPECIFICATION
00420 08 *** INVALID MODE SPECIFICATION
00430 08 *** LABEL MUST BE SPECIFIED
00440 08 *** INVALID MODE SPECIFICATION
00450 08 *** MORE THAN ONE LOCAL 'ATTNLIST' HAS BEEN CODED
00460 08 *** MORE THAN ONE GLOBAL 'ATTNLIST' HAS BEEN CODED
00470 08 *** ATTNLIST: SCOPE= MUST BE 'LOCAL' OR 'GLOBAL'
00480 08 *** ILLEGAL NUMBER OF OPERANDS - MUST BE EVEN
00490 08 *** ATTNLIST COMMAND STRING MUST BE 1-8 CHARACTERS IN LENGTH
00500 08 *** NO ACTIVE 'IF' OR 'DO' STRUCTURE
00510 08 *** OPERAND IS NOT 'GOTO' OR 'THEN'
00520 08 *** IF/DO NESTING LIMIT EXCEEDED
00530 08 *** INVALID CONJUNCTION SPECIFIED (MUST BE 'AND' OR 'OR')
00540 08 *** INVALID RELATIONAL OPERATOR SPECIFIED
00550 08 *** CONDITION MUST BE 'EQ' OR 'NE' FOR 'STRING COMPARE'
00560 08 *** ACTIVE STRUCTURE IS NOT 'IF'
00570 08 *** 'DO WHILE' OR 'DO UNTIL' MUST HAVE EVEN NUMBER OF OPERANDS
00580 08 *** ACTIVE STRUCTURE IS NOT 'DO'
00590 08 *** AN 'IF/ELSE/ENDIF' OR 'DO/ENDDO' CLAUSE HAS NOT BEEN TERMINATED
00600 08 *** ERROR 60 (RESERVED FOR 'DO')
00610 08 *** SPECIFY 'WAIT=YES' OR 'WAIT=NO' FOR DISK OPERATIONS
00620 08 *** IF 'WAIT=NO', 'ERROR=' AND 'END=' MAY NOT BE SPECIFIED
00630 08 *** UNBALANCED QUOTES IN OPERAND
00640 08 *** INVALID PROMPT MESSAGE
00650 08 *** 'COUNT' MUST BE A POSITIVE SELF-DEFINING TERM
00660 08 *** INVALID DATA TYPE SPECIFIED
00670 08 *** 'COUNT' MAY NOT BE MORE THAN 2 WITH REGISTER OPERANDS
00680 08 *** DATA TYPE MUST BE 'WORD' WITH REGISTER OPERANDS
00690 08 *** 'RESULT=' MAY NOT BE SPECIFIED WITH 'MOVE' OR 'MOVEA'
00700 08 *** INVALID 'BUSY' SPECIFICATION
00710 08 *** SECOND OPERAND NOT 'RESET' OR 'CLEAR'
00720 08 *** NO OTHER OPERANDS ALLOWED WITH 'TIMER' OR 'ENTER' WAIT
00730 08 *** REGISTER SPECIFICATION INVALID
00740 08 *** INVALID RESOURCE SPECIFICATION
00750 08 *** 'CODE' MUST BE SELF-DEFINING TERM
00760 08 *** 'NLINES' MUST BE POS., SELF-DEFINING TERM
00770 08 *** 'NLINES' REQUIRED WITH 'NSPACES' SPECIFICATION
00780 08 *** 'NSPACES' MUST BE POS., SELF-DEFINING TERM
00790 08 *** INVALID OPERAND SPECIFIED ON 'TERMCTRL'
00800 08 *** INVALID 'TYPE=', MUST BE 'DATA' OR 'ALL'
00810 08 *** INVALID 'MODE=', MUST BE 'FIELD', 'LINE', OR 'SCREEN'
00820 08 *** INVALID FORMAT IN OPERAND 1
00830 08 *** NO CHARACTER STRING SPECIFIED
00840 08 *** OPERAND 3 IS MISSING
00850 08 *** INCOMPATIBLE MARGINS
00860 08 *** INVALID SPECIFICATION FOR 'SCREEN'
00870 08 *** INVALID SPECIFICATION FOR 'OVFLINE'
00880 08 *** NO STORAGE ADDRESS SPECIFIED
00890 08 *** NO BRANCH ADDRESS SPECIFIED
00900 08 *** INVALID SENSOR INPUT/OUTPUT TYPE
00910 08 *** INVALID 'ERROR=' SPECIFIED
00920 08 *** 'BITS=' INVALID FOR 'AI' AND 'AO'
00930 08 *** INVALID 'SEQ=' FOR 'AI'
00940 08 *** INVALID 'BITS=', MUST HAVE THE FORM 'BITS=(U,V)'
00950 08 *** INVALID 'LS3' SPECIFIED
00960 08 *** INVALID 'PULSE' SPECIFICATION
00970 08 *** INVALID 'EOB' SPECIFIED
00980 08 *** INVALID 'TERMINAL NAME', MUST BE 1-8 CHARACTERS
00990 08 *** INVALID HEXADECIMAL CONSTANT SPECIFIED
01000 08 *** NEITHER POSITIONAL NOR KEYWORD PARAMETERS WERE SPECIFIED
01010 08 *** A DATA ADDRESS MUST BE SPECIFIED
01020 08 *** INVALID OR UNSPECIFIED LENGTH OPERAND
01030 08 *** OTE TYPE MUST BE SPECIFIED
01040 08 *** INVALID DUPLICATION FACTOR
01050 08 *** INVALID 'FORMAT=' SPECIFICATION
01060 08 *** DATA TYPE MUST BE 'WORD' OR 'BYTE'
01070 08 *** ILLEGAL CONTINUATION - DATA MUST START IN COLUMN 16
01080 08 *** 'BITS=' MUST BE SPECIFIED WITH 'TYPE=SUBGROUP'
01090 08 *** PCB NOT SPECIFIED
01100 08 *** INVALID 'ADDRESS=', MUST BE BETWEEN '00' AND 'FF'
01110 08 *** INVALID 'TYPE=' SPECIFIED
01120 08 *** INVALID 'BIT=', MUST BE BETWEEN '0' AND '15'
01130 08 *** 'SPECPI=' MUST BE SPECIFIED FOR 'TYPE=GROUP' AND 'TYPE=BIT'

```

Figure B-3. \$EDXL listing (2 of 4)

```

01140 08 *** INVALID 'POINT=', MUST BE '0-15' FOR AI OR '0-1' FOR AO
01150 08 *** 'ADC' ADDRESS SPECIFIED INSTEAD OF 'MULTIPLEXER' ADDRESS
01160 08 *** INVALID 'RANGE=', MUST BE 5V,500MV,200MV,100MV,50MV,20MV,OR,10MV
01170 08 *** INVALID 'ZCOR=', MUST BE 'YES' OR 'NO'
01180 08 *** INVALID OR MISSING COUNT= SPECIFICATION
01190 08 *** INVALID OR MISSING SIZE= SPECIFICATION
01200 08 *** INVALID 'LOGMSG=', MUST BE 'YES' OR 'NO'
01210 08 *** INVALID 'DS=' ON LOAD
01220 08 *** INVALID 'DS=' ON OVERLAY LOAD, MUST HAVE THE FORM 'DSX'
01230 08 *** NO OPEN 'TASK' STATEMENT FOR THIS 'ENDTASK'
01240 08 *** TYPE COUNT MUST BE BETWEEN 0 AND 255
01250 08 *** INVALID GPIB OPERATION
01260 08 *** INDEX REGISTER IS AN INVALID OPERAND
01270 08 *** INVALID FIRST CHARACTER IN PREC=
01280 08 *** INVALID SECOND CHARACTER IN PREC=
01290 08 *** INVALID THIRD CHARACTER IN PREC=
01300 08 *** MAXIMUM OF 3 PREC= SPECIFICATIONS
01310 08 *** INVALID COUNT= PARAMETER
01320 08 *** INVALID PRECISION FOR IMMEDIATE OPERAND 2
01330 08 *** INVALID DATA TYPE COMBINATION
01340 08 *** TOO FEW PREC= SPECIFICATIONS
01350 08 *** INVALID FORMAT= SPECIFICATION
01360 08 *** MAXIMUM OF 8 HEXADECIMAL DIGITS (4 BYTES) PER OPERAND
01370 08 *** DATA TYPE SPECIFICATION IS NOT RECOGNIZED
01380 08 *** FLOATING POINT CONVERSION ERROR OR EBFLCVT NOT IN SUPERVISOR
01390 08 *** INVALID KEYWORD COMBINATION
01400 08 *** STORAGE SIZE MUST BE SPECIFIED (16K - 256K)
01410 08 *** MAX. NUMBER OF PROGRAMS NOT BETWEEN 1 AND 100
01420 08 *** INVALID TP= SPECIFICATION
01430 08 *** MAXPROG= AND PARTS= DO NOT MATCH
01440 08 *** PARTITION SIZE EXCEEDS 32 BLOCKS
01450 08 *** INVALID DISK= OPERAND
01460 08 *** OUT OF SEQUENCE, END=YES PREVIOUSLY SPECIFIED
01470 08 *** TYPE=DSECT IS NOT SUPPORTED
01480 08 *** INVALID OR MISSING DEVICE TYPE SPECIFIED
01490 08 *** A DEVICE ADDRESS MUST BE SPECIFIED
01500 08 *** DEVICE ADDRESS MUST BE FROM X'00' TO X'FF'
01510 08 *** VOLUME LABEL MUST BE SPECIFIED
01520 08 *** VOLUME LABEL IS MORE THAN 6 CHARACTERS
01530 08 *** INVALID LIBRARY ORIGIN SPECIFICATION
01540 08 *** INVALID OR MISSING VOLUME ORIGIN SPECIFICATION
01550 08 *** INVALID OR MISSING VOLUME SIZE SPECIFICATION
01560 08 *** INVALID OR MISSING FIXED HEAD VOLUME SPECIFICATION
01570 08 *** SECONDARY VOLUMES NOT ALLOWED FOR 4964
01580 08 *** RECORDS PER VOLUME EXCEEDS 32760
01590 08 *** COUNT TOO HIGH IN 'PARM=' OPERAND
01600 08 *** ONLY 1 HOSTCOMM STATEMENT IS ALLOWED
01610 08 *** INCONSISTENT TOP MARGIN
01620 08 *** INCONSISTENT BOTTOM MARGIN
01630 08 *** INVALID LEVEL SPECIFICATION
01640 08 *** TOO MANY PI= ENTRIES
01650 08 *** INVALID SPECIFICATION FOR ECHO
01660 08 *** STATIC SCREENS ARE NOT SUPPORTED FOR THIS TERMINAL TYPE
01670 08 *** THE SECOND PI ENTRY IS INVALID
01680 08 *** THE TWO PI ENTRIES ARE EQUAL
01690 08 *** THE FIRST PI ENTRY IS INVALID
01700 08 *** THIS ADDRESS HAS BEEN PREVIOUSLY DEFINED
01710 08 *** INVALID AITYPE=
01720 08 *** INVALID 4982 FEATURE ADDRESS
01730 08 *** INVALID 4982 BASE ADDRESS
01740 08 *** REQUIRED PARAMETER IS MISSING
01750 08 *** SCAN= PARAMETER IS INCORRECT
01760 08 *** ACTION= PARAMETER IS INCORRECT
01770 08 *** INVALID PARAMETER IN DATA LIST
01780 08 *** FORMAT SPECIFICATION IS INVALID
01790 08 *** FORMAT - CONVERT SPECIFICATION IS INVALID
01800 08 *** FORMAT - PARENS SPECIFICATION IS INVALID
01810 08 *** FORMAT - DELIMITER SPECIFICATION IS INVALID
01820 08 *** FORMAT - X-TYPE SPECIFICATION IS INVALID
01830 08 *** FORMAT - F-TYPE SPECIFICATION IS INVALID
01840 08 *** FORMAT - I-TYPE SPECIFICATION IS INVALID
01850 08 *** FORMAT - A-TYPE SPECIFICATION IS INVALID
01860 08 *** FORMAT - NUMERIC SPECIFICATION IS INVALID
01870 08 *** FORMAT - H-TYPE SPECIFICATION IS INVALID
01880 08 *** FORMAT - /-TYPE SPECIFICATION IS INVALID
01890 08 *** FORMAT - LIST SPECIFICATION IS INVALID
01900 08 *** FORMAT - EXCEEDS MAXIMUM NUMBER OF SPECIFICATIONS (40)
01910 08 *** FORMAT - MAXIMUM CHARACTER STRING IS 254
01920 08 *** INVALID BSCREAD/BSCWRITE TYPE SPECIFICATION
01930 08 *** INVALID TIMEOUT OPERAND
01940 08 *** INVALID ADDRESS OPERAND
01950 08 *** INVALID RETRIES OPERAND
01960 08 *** INVALID MC OPERAND
01970 08 *** INVALID TYPE OPERAND
01980 08 *** INVALID BSCIOCB ADDRESS SPECIFICATION
01990 08 *** THE TOTAL NUMBER OF OPERAND DELIMITERS EXCEEDS MAXIMUM (50)
02000 08 *** INSUFFICIENT STORAGE AVAILABLE FOR TERMINAL PROCESSING
02010 08 *** LOADER ERROR WHILE PROCESSING TERMINAL STATEMENT
02020 08 *** COUNT NOT BETWEEN 0 AND 32767

```

Figure B-3. \$EDXL listing (3 of 4)

```

02030 08 *** FORMAT SPECIFICATION NOT ALLOWED WITHIN GET/PUTEDIT
02040 08 *** INVALID BIT RATE/RANGE SPECIFICATION
02050 09 *** MUST HAVE LINEDEL OR CR OR ATTN OR COD SPECIFIED
02060 08 *** CRDELAY SPECIFIED INCORRECTLY
02070 08 *** NAME SUBLIST .GT. PARM SUBLIST
02080 08 *** PART NOT ALLOWED WITH PARAMETERS, EVENT= OR OVLY PROGRAMS
02090 08 *** PART NOT ALLOWED WITH START= OR LOADPT=
02100 08 *** PART NOT ALLOWED WITH DSX SPECIFICATIONS
02110 08 *** INVALID IMMEDIATE OPERAND IN STRING COMPARE
02120 08 *** INVALID COPYCODE LIBRARY NAME
02130 0P *** DISK I/O ERROR DURING OPEN OF COPYCODE DATA SET
02140 08 *** DATA SET NAME $$$--- NOT PERMITTED FOR COPYCODE
02150 08 *** SPECIFIED COPYCODE MODULE IS NOT A DATA SET
02160 08 *** 'COMMAND=' MUST BE SPECIFIED
02170 08 *** 'ADDRESS=' MUST BE SPECIFIED
02180 08 *** INVALID 'COMMAND='
02190 08 *** 'LEVEL' MUST BE EITHER 0, 1, 2, OR 3
02200 08 *** 'IBIT' MUST BE EITHER 0 OR 1
02210 08 *** INVALID HEXADECIMAL ENTRY
02220 08 *** 'DCB' ADDRESS MUST BE SPECIFIED
02230 08 *** 'MOD4' MUST BE SPECIFIED
02240 08 *** 'DEVMOD=' MUST BE SPECIFIED
02250 08 *** 'IOTYPE=' MUST BE 'INPUT' OR 'OUTPUT'
02260 08 *** 'DATAADDR=' MUST BE SPECIFIED
02270 08 *** 'CHAINAD=' MUST BE SPECIFIED
02280 08 *** INVALID 'END=' MUST BE 'YES' OR 'NO'
02290 08 *** 'MAXDCB=' OUT OF LIMITS
02300 08 *** 'RSd=' MUST BE EVEN
02310 08 *** 'RSB=' OUT OF LIMITS
02320 08 *** 'PCI=' MUST BE 'YES' OR 'NO'
02330 08 *** 'XD=' MUST BE 'YES' OR 'NO'
02340 08 *** 'SE=' MUST BE 'YES' OR 'NO'
02350 08 *** 'DEVADDR ' POSITIONAL PARAMETER MISSING
02360 08 *** 'ECBADDR ' POSITIONAL PARAMETER MISSING
02370 08 *** 'IOCBADDR ' POSITIONAL PARAMETER MISSING
02380 08 *** INVALID NUMERIC OPERAND
02390 08 *** TO KEY SPECIFIED WITH #1 OR #2
02400 08 *** FROM KEY SPECIFIED WITH #1 OR #2
02410 08 *** INVALID PRECISION SPECIFIED WITH FROM KEY OR TO KEY
02420 08 *** FROM KEY SPECIFIED WITH IMMEDIATE OPERAND
02430 08 *** #1 OR #2 USED IN FROM KEY OR TO KEY
02440 08 *** IA BUFFER LENGTH NOT BETWEEN 10 AND 100
02450 08 *** INVALID 'ADAPTER=' OPERAND CODED
02460 08 *** INVALID VCLJME LABEL ON *COPYCOD RECORD IN $FDXL
02470 08 *** OPERAND FIELD LENGTH EXCEEDS 254 CHARACTERS
02480 08 *** ID= MUST BE SPECIFIED, AND A UNIQUE 1-6 CHARACTER LABEL
02490 08 *** LABEL= MUST BE EITHER SL, NL, OR BLP (DEFAULT=SL)
02500 08 *** DENSITY= MUST BE EITHER 800, 1600, OR DUAL (DEFAULT=1600)
02510 08 *** INVALID 'INITPART' OR 'INITMOD' PARM ON 'SYSTEM'
02520 08 *** INVALID TCB LABEL ON 'TCBGET' OR 'TCBPUT'
02530 *OVERLAY $ASM0006 ASMLIB IF DO ELSE ENDF ENDDO
02540 MOVE MOVEA AND IDR EOR SHIFL SHIFTR
02550 *OVERLAY $ASM0001 ASMLIB ENQT DEQT COPY USER SQRT
02560 *COMMENT
02570 *OVERLAY $ASM0002 ASMLIB ADD DIVIDE MULTIPLY *MULT SUBTRACT
02580 SUB GOTO RESET TIMER RETURN INTIME GETTIME ADDV
02590 *COMMENT
02600 *OVERLAY $ASM0003 ASMLIB PROGRAM LOAD DSCB
02610 *COMMENT
02620 *OVERLAY $ASM0004 ASMLIB PRINDATE PRINTIME QUESTION TEXT ERASE
02630 RDCURSOR TERMCTRL HASHVAL
02640 *OVERLAY $ASM0005 ASMLIB ENDPROG ENDTASK PROGSTOP TASK ATTACH
02650 DETACH ATTNLIST ENDATTN
02660 *OVERLAY $ASM0007 ASMLIB DC EQU DATA ECB QCB
02670 BUFFER DS IOCB EXTRN WXRN ENTRY CSECT
02680 *OVERLAY $ASM0008 ASMLIB READ WRITE NOTE POINT CONTROL
02690 *OVERLAY $ASM0009 ASMLIB WAIT POST ENQ DEQ CALL
02700 SUBROUT CALLFORT
02710 *OVERLAY $ASM000A ASMLIB GETEDIT PUTEDIT
02720 *OVERLAY $ASM000B ASMLIB SBIO IODEF
02730 *OVERLAY $ASM000C ASMLIB FIND FINDNOT
02740 *OVERLAY $ASM000D ASMLIB FPCONV FADD FSUB FMULT FDIVD
02750 *OVERLAY $ASM000E ASMLIB PRINTNUM GETVALUE READTEXT PRINTTEXT CONVRTB
02760 CONVRTD
02770 *OVERLAY $ASM000G ASMLIB PLOTGIN GIN SCREEN XYPLOT YTPLOT
02780 CONCAT TP STATUS
02790 *OVERLAY $ASM000H ASMLIB BSCREAD BSCWRITE BSCOPEN BSCCLOSE BSCIOCB
02800 BSCLINE
02810 *OVERLAY $ASM000I ASMLIB FORMAT
02820 *OVERLAY $ASM000Q ASMLIB FIRSTQ LASTQ NEXTQ DEFINEQ
02830 *OVERLAY $ASMEXIO ASMLIB EXIODEV IOCB DCB EXOPEN EXIO
02840 *OVERLAY $ASM000S ASMLIB SYSTEM STOREMAP DISK TIMER TAPE
02850 *OVERLAY $ASM000T ASMLIB TERMINAL
02860 *OVERLAY $ASM000U ASMLIB HOSTCOMM SENSORIO DD$SIO GETMAIN FREEMAIN
02870 *OVERLAY $ASM000F ASMLIB ASMERROR $IDFQ OTE SLE
02880 *OVERLAY $ASM000M ASMLIB WHERE$ TCBGET TCBPUT
02890 *COPYCOD ASMLIB
02900 *COPYCOD EDX002
02910 **STOP**

```

Figure B-3. \$EDXL listing (4 of 4)

```

00010 JOB          STATIC
00020 LOG           $SYSPRTR
00030 *
00040 * THIS ASSEMBLY USES A COPY CODE MODULE NAMED ROLL
00050 * ON EDX003. THE *COPYCODE DEFINITION STATEMENT
00060 * DEFINING EDX003 AS THE COPYCODE VOLUME IS IN A
00070 * USER DEFINED LANGUAGE CONTROL DATA SET NAMED
00080 * 'STATEDXL'. 'STATEDXL' IS A COPY OF THE SYSTEM
00090 * SUPPLIED LANGUAGE CONTROL DATA SET '$EDXL',
00100 * WITH THE *COPYCODE STATEMENT FOR EDX003 ADDED.
00110 *
00120 PROGRAM $EDXASM,ASMLIB
00130 REMARK ASSEMBLY OF 'STATSRC' STARTED
00140 DS      STATSRC
00150 DS      ASMWORK
00160 DS      ASMOUT
00170 PARM   LIST          $SYSPRTR          STATEDXL
00180 NOMSG
00190 EXEC
00200 JUMP   BADASM,NE,-1
00210 *
00220 * THIS LINK INCLUDES THE '$IM' SUBROUTINE SUPPORT BY
00230 * USE OF THE AUTOCALL OPTION.THE AUTOCALL DEFINITION
00240 * STATEMENTS FOR THE '$IM' SUPPORT ARE IN THE SYSTEM
00250 * SUPPLIED AUTOCALL DATA SET '$AUTO' ON ASMLIB.
00260 *
00270 PROGRAM $LINK,EDX002
00280 REMARK LINK EDIT OF 'ASMOUT' OBJECT MODULE STARTED
00290 REMARK NAME OF LINK CONTROL DATA SET ?
00300 PAUSE
00310 DS      LINKWRK1
00320 DS      LINKWRK2
00330 PARM   $SYSPRTR
00340 NOMSG
00350 EXEC
00360 JUMP   BADLINK,NE,-1
00370 PROC   FORMPROC
00380 JUMP   END,EQ,-1
00390 REMARK FORMAT STEP FAILED
00400 JUMP   END
00410 LABEL BADASM
00420 REMARK ASSEMBLY STEP FAILED
00430 JUMP   END
00440 LABEL BADLINK
00450 REMARK LINK EDIT STEP FAILED
00460 LABEL END
00470 EOJ

```

Figure B-4. \$JOBUTIL listing

```

LOG           $SYSPRTR
PROGRAM $EDXASM,ASMLIB
DS          STATSRC
DS          ASMWORK
DS          ASMOUT
PARM       LIST          $SYSPRTR          STATEDXL
NOMSG
EXEC

```

EDX ASSEMBLER STATISTICS

```

SOURCE INPUT - STATSRC ,EDX002
WORK DATA SET - ASMWORK ,EDX002
OBJECT MODULE - ASMOUT ,EDX002
STATEMENTS PROCESSED - 77

```

NO STATEMENTS FLAGGED

LOC	+0	+2	+4	+6	+8	SOURCE STATEMENT	STATSRC ,EDX002	(5719-XX4)-V3.0.0
0000	0008	0709	D6C7	D9C1	D440	XMPLSTAT PROGRAM START		00000010
000A	0000	05E4	0370	0000	0000			
0014	06E6	0000	0000	0000	0100			
001E	06E4	0000	0000	0000	0664			
0028	0000	0000	0000	0000	0000			
0032	0000							
0034						EXTRN \$IMOPEN,\$IMDEFN,\$IMPROT,\$IMDATA		00000020

Figure B-5. STATPROC execution output (1 of 4)

```

0034 0000 0300 0000 0000 0000 IMAGEBUF BUFFER 768,BYTES 00000030
003F 0000 0000 0000 0000 0000
0336 0000
0338 0E0D E5C9 C4C5 D6F1 68C1 DSETNAME TEXT *VIDE01,ASMVOL* 00000040
0342 E2D4 E5D6 D340
0348 4040 4040 4040 4040 8000 IOCB1 IOCB NHIST=0 00000050
0352 00FF 0000 7FFF 0000 0000
035C 4040 4040 4040 4040 8800 IOCB2 IOCB SCREEN=STATIC 00000060
0366 00FF 0000 7FFF 0000 0000
0370 0002 0403 C5D5 C440 05B2 ATTNLIST (END,OUT,$PF,STATIC) 00000070
037A 0403 5B07 C640 05BA COPY ROLL 00000130
*
*START OF "COPYCODE" MODULE
0382 9025 0348 *START ENQT IOCB1 00000001
0386 B02A 0001 000F 8026 1414 PRINTTEXT *CLASS ROSTER PROGRAM',SPACES=15,LINE=1 00000002
0390 C3D3 C1E2 E240 D9D6 E2E3 00000003
039A C5D9 40D7 D9D6 C7D9 C1D4 00000130
03A4 902A 0902 0000 8026 2020 PRINTTEXT *HIT ''ATTN'' AND ENTER ''END'' TO END',SKIP=2 00000150
03AE C8C9 E340 7DC1 E3E3 D57D
03B8 40C1 D5C4 40C5 D5E3 C5D9
03C2 407D C505 C47D E306 40C5
03CC D5C4
03CE 8026 0C0B E3C8 C540 D7D9 PRINTTEXT *THE PROGRAM* 00000160
03D8 D6C7 D9C1 D440
03DE 902A 0002 0000 8026 201F PRINTTEXT *HIT ANY PROGRAM FUNCTION KEY TO',SKIP=2 00000170
03EB C8C9 E340 C1D5 E840 D7D9
03F2 D6C7 D9C1 D440 C6E4 D5C3
03FC E3C9 D605 40D2 C5E8 40E3
0406 D640
0408 8026 1A1A 40C2 D9C9 D5C7 PRINTTEXT * BRING UP THE ENTRY SCREEN' 00000180
0412 40E4 D740 E3C8 C540 C5D5
041C E3D9 E840 E2C3 D9C5 C5D5
0426 00B2 DEQT 00000190
*
* END OF "COPYCODE" MODULE
00000200
00000210
00000220
0428 0018 05C2 *CHECK WAIT ATTNECB,RESET 00000240
042C A0A2 05C2 0001 055C IF (ATTNECB,EQ,1),GOTO,ENDIT 00000250
0434 C29E 0000 033A 0038 GETIMAGE CALL $IMOPEN,(DSETNAME),(IMAGEBUF) 00000260
043C A0A2 05E6 FFFF 0472 IF (XMPLSTAT+2,NE,-1) 00000270
0444 0J5C 05CA 05E6 MOVE ERRCODE,XMPLSTAT+2 00000280
044A 8026 1A1A 7CC9 D4C1 C7C5 PRINTTEXT *%IMAGE OPEN ERROR, CODE = ' 00000290
0454 40D6 D7C5 D540 C5D9 D9D6
045E D96B 40C3 D6C4 C540 7E40
0468 0028 05CA 0001 PRINTNUM ERRCODE 00000300
046E 00A0 05CC GOTO ERRQUERY 00000310
0472 ENDIF 00000320
047E C29E 0000 0348 0038 CALL $IMDEFN,(IOCB1),(IMAGEBUF) 00000330
047A 9025 035C ENQT IOCB2 00000340
047E 1430 TERMCTRL BLANK 00000350
0480 C29E 0000 0038 0000 CALL $IMPROT,(IMAGEBUF),0 00000360
0488 B19E 0030 0038 CALL $IMDATA,(IMAGEBUF) 00000370
048E B02A 0004 000B PRINTTEXT LINE=4,SPACES=11 00000380
0494 1C30 TERMCTRL DISPLAY 00000390
0496 2030 WAIT KEY 00000400
0498 00A1 05E6 0004 050E 04AB WAITONE GOTO (READ,E1,E2,E3,E4),XMPLSTAT+2 00000410
04A2 04B2 04B0 04C6
04AB 805C 05C8 0006 E1 MOVE LINENBR,6 00000420
04AE 00A0 04CC GOTO DELETE 00000430
04B2 805C 05C8 000B E2 MOVE LINENBR,11 00000440
04B8 00A0 04CC GOTO DELETE 00000450
04B0 805C 05C8 0010 E3 MOVE LINENBR,16 00000460
04C2 00A0 04CC GOTO DELETE 00000470
04C6 805C 05C8 0015 E4 MOVE LINENBR,21 00000480
04CC E02A 05C8 0000 F030 0004 DELETE ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR 00000490
04D6 2000
04D8 8032 05C8 0001 ADD LINENBR,1 00000500
04DE E02A 05C8 0000 F030 0004 ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR 00000510
04E8 2000
04EA 8032 05C8 0001 ADD LINENBR,1 00000520
04FO E02A 05C8 0000 F030 0004 ERASE MODE=LINE,TYPE=DATA,LINE=LINENBR 00000530
04FA 2000
04FC 8035 05C8 0002 SUBTRACT LINENBR,2 00000540
0502 A02A 05C8 0005 PRINTTEXT LINE=LINENBR,SPACES=5 00000550
0508 1C30 TERMCTRL DISPLAY 00000560
050A 00A0 0496 GOTO WAITONE 00000570
050E F02A 0002 0037 C026 100F READ QUESTION *MORE ENTRIES ? ',LINE=2,SPACES=55,NO=CLEANUP 00000580
0518 D4D6 D9C5 40C5 D5E3 D9C9
0522 C5E2 406F 4040 802E 0550
052C F02A 0002 0037 F030 0004 ERASE MODE=LINE,LINE=2,SPACES=55,TYPE=DATA 00000590
0536 2000
0538 F02A 0006 0000 F030 0000 ERASE MODE=SCREEN,LINE=6 00000600
0542 2000
0544 F02A 0006 0005 PRINTTEXT LINE=6,SPACES=5 00000610
054A 1C30 TERMCTRL DISPLAY 00000620
054C 00A0 0496 GOTO WAITONE 00000630
0550 F030 0001 2000 CLEANUP ERASE MODE=SCREEN,TYPE=ALL 00000640

```

Figure B-5. STATPROC execution output (2 of 4)

```

0556 00B2 DEQT 00000650
0558 00A0 0382 GOTO START 00000660
055C 0022 FFFF PROGSTOP 00000670
0560 5050 DATA X'5050' 00000680
0562 6060 6060 6060 6060 6060 DASHES DATA 80C'- ' 00000690
0582 0019 05C2 0001 OUT POST ATTNECB,1 00000700
0588 001D ENDATTN 00000710
05BA 0019 05C2 FFFF STATIC POST ATTNECB,-1 00000720
05C0 001D ENDATTN 00000730
05C2 FFFF 0000 0000 ATTNECB ECB 00000740
05C8 0000 LINENBR DATA F'0' 00000750
05CA 0000 ERRCODE DATA F'0' 00000760
05CC C026 0E3E 7CD9 C5E3 D9E8 ERRQUERY QUESTION '3RETRY OPEN ? ',YES=GETIMAGE,NO=ENDIT 00000770
05D6 40D6 D7C5 D540 6F40 C02E
05E0 0434 055C
05E4 0000 0000 0000 0234 0000 ENDPROG 00000780
05EE 0000 0000 0382 05E4 0000
05F8 0000 0000 0000 0000 0000
0602 0002 0096 0300 0000 FFFF
060C 0000 0000 0610 0000 0000
0616 0612 E7D4 D7D3 E2E3 C1E3
0620 0000 0000 0000 0000 0000
062A 0000 0000 FFFF 0000 0000
0634 0000 0000 0000 05E4 0000
063E 0000 0000 0000 0000 0000
065C 0000 0000 05E4 0000 0000
0666 0000 0000 0234 0000 0000
0670 0000 0000 0664 0000 0000
067A 0000 0000 0000 0000 0001
0684 000A 0000 0000 FFFF 0000
068F 0000 0690 0000 0000 0692
0698 5BC1 E3E3 C1E2 D240 0000
06A2 0000 0000 0000 0000 0000
06AC 0000 FFFF 0000 0000 0000
06B6 0000 0000 05E4 0000 0000
06C0 0000 0000 0000 0000 0000
06DE 0000 0664 0080 0000 0000
06E8 0000 0000 0000 0000 0000
06F2 0000 0000 0000 0000
06FA

```

END  
EXTERNAL/UNDEFINED SYMBOLS

```

SVC      WXTRN
SUPEXIT  WXTRN
SETBUSY  WXTRN
$IMOPEN  EXTRN
$IMDEFN  EXTRN
$IMPROT  EXTRN
$IMDATA  EXTRN

```

```

COMPLETION CODE = -1
JUMP      BADASM,NE,-1
PROGRAM   $LINK,EDX002
DS        LINKSTAT
DS        LINKWRK1
DS        LINKWRK2
PARM     $SYSRTR
NOMSG
EXEC
$LINK EXECUTION CONTROL RECORDS
FROM LINKSTAT,EDX002
* THIS LINK CONTROL DATA SET SPECIFIES:
* 1.) THE LINKED OUTPUT OBJECT MODULE WILL
*    BE STORED IN 'LINKOUT' ON EDX002
* 2.) THE AUTOCALL DATA SET IS '$AUTO' ON
*    ASMLIB (SYSTEM SUPPLIED)
* 3.) '$ASMOU' ON EDX002 IS THE ONLY INPUT
*    OBJECT MODULE TO BE INCLUDED
OUTPUT LINKOUT AUTO=$AUTO,ASMLIB
INCLUDE ASMOU
INCLUDE $IMOPEN,ASMLIB VIA AUTOCALL
INCLUDE $IMGEN,ASMLIB VIA AUTOCALL
INCLUDE $IMTYPE,ASMLIB VIA AUTOCALL
INCLUDE $$RETURN,ASMLIB VIA AUTOCALL
INCLUDE $UNPACK,ASMLIB VIA AUTOCALL
END

```

Figure B-5. STATPROC execution output (3 of 4)

\*\*\*\*\* UNRESOLVED EXTERNAL REFERENCES

```

WXTRN  SVC
WXTRN  SUPEXIT
WXTRN  SETBUSY
OUTPUT NAME= LINKOUT
ESD TYPE LABEL ADDR LENGTH
CSECT          0000      06FA
CSECT          06FA      09E8
  ENTRY $IMOPEN 06FC
  ENTRY $FILE   0908
  ENTRY DISKBUFR 095C
  ENTRY DSOPEM  0A7A
  ENTRY $DSNFND 0FE2
  ENTRY $DSBIODA 0FE4
  ENTRY $DSBVOL 0FE6
  ENTRY $DSBLIB 0FE8
  ENTRY $DSIOERR 0FEA
  ENTRY $DSNVTOC 0FEC
  ENTRY $$EXIT  0FEE
CSECT          10E2      0E2A
  ENTRY $IMDEFN 10E4
  ENTRY $IMPROT 1180
  ENTRY $IMDATA 1908
  ENTRY $ADDRTBL 1EA8
  ENTRY $ATTRTBL 1EF8
CSECT          1F0C      0074
  ENTRY $IMDTYPE 1F0E
CSECT          1F80      0028
  ENTRY RETURN  1F80
CSECT          1FA8      0040
  ENTRY $UNPACK 1FAA
MODULE TEXT LENGTH= 1FE8, RLD COUNT= 1015
LINKOUT ADDED TO EDX002

```

\$LINK COMPLETION CODE= -1

```

$LINK ENDED
JUMP BADLINK,NE,-1
PROGRAM $UPDATE
  PARM $SYSPRTR LINKOUT STATPROG YES
NOMSG
EXEC
STATPROG STORED

```

```

$UPDATE ENDED
JUMP END,EQ,-1
LABEL END

```

Figure B-5. STATPROC execution output (4 of 4)

# READER'S COMMENT FORM

SR30-0436-1

IBM Series/1  
Event Driven Executive  
Licensed Program  
Study Guide

Please use this form *only* to identify publications errors or request changes to publications. Inaccurate or misleading information in this publication may be corrected by your comments.

Technical questions or suggestions about IBM systems, programming changes or requests for additional publications should be directed to your IBM branch office.

List specific errors, omissions, suggestions, additions, and deletions by page number in the space provided.

---

## COMMENTS

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

If you wish a reply, be sure to print your name and address:

Name \_\_\_\_\_

Address \_\_\_\_\_

Zip Code \_\_\_\_\_

IBM Branch office: No. \_\_\_\_\_ City \_\_\_\_\_

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.  
FOLD ON TWO LINES, SEAL AND MAIL

Fold and tape

Please Do Not Staple

Fold and tape

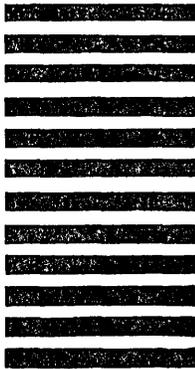


NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation  
Technical Publications, Dept. 796  
P. O. Box 2150  
Atlanta, Georgia 30055



Fold and tape

Please Do Not Staple

Fold and tape

# READER'S COMMENT FORM

SR30-0436-1

IBM Series/1  
Event Driven Executive  
Licensed Program  
Study Guide

Please use this form *only* to identify publications errors or request changes to publications. Inaccurate or misleading information in this publication may be corrected by your comments.

Technical questions or suggestions about IBM systems, programming changes or requests for additional publications should be directed to your IBM branch office.

List specific errors, omissions, suggestions, additions, and deletions by page number in the space provided.

---

## COMMENTS

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

If you wish a reply, be sure to print your name and address:

Name \_\_\_\_\_

Address \_\_\_\_\_

Zip Code \_\_\_\_\_

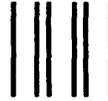
IBM Branch office: No. \_\_\_\_\_ City \_\_\_\_\_

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.  
FOLD ON TWO LINES, SEAL AND MAIL

Fold and tape

Please Do Not Staple

Fold and tape



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation  
Technical Publications, Dept. 796  
P. O. Box 2150  
Atlanta, Georgia 30055



Fold and tape

Please Do Not Staple

Fold and tape

**IBM**®