IBM

Series/1

SC34-0635-0

Event Driven Executive Customization Guide

Version 5.0

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

IBM

Series/1

SC34-0635-0

Event Driven Executive Customization Guide

Version 5.0

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

First Edition (December 1984)

Summary of Changes for Version 5.0

The following additions and changes have been made to this document:

- *Chapter 3, Customizing the Session Manager* has been updated to include changes to the Session Manager screens.
- *Chapter 8, Techniques for Improving Performance* has been updated to include new performance tips using the $MEMDISK utility.

About This Book

This book describes how to extend or enhance some of the EDX software facilities to meet your own requirements. None of the modifications you make are required to use the facilities as distributed.

Audience

This book is intended for application programmers who write and maintain programs using the Event Driven Language. Readers should be familiar with the language before using this book. You can learn the Event Driven Language by using the *Event Driven Executive Language Programming Guide*.

The *Internal Design* can assist you in understanding some of the topics presented. Other topics will require you to be familiar with assembler language programming and hardware control blocks.

About This Book

How This Book Is Organized

This book contains eight chapters:

- *Chapter 1. What is Customization* overviews the facilities you can enhance or extend and presents some ideas you could implement.
- *Chapter 2. Adding Your Own Operator Command* describes how to create a new operator command. It explains design considerations and several coding examples.
- *Chapter 3. Customizing the Session Manager* shows how to add options and build menus that run under the session manager. This chapter also presents several different techniques you can use when adding an option.
- *Chapter 4. Adding Your Own Task Error Exit Routine* describes how you can pass control from a main program to an error-handling routine when a program check occurs.
- *Chapter 5. Running Programs and Initialization Routines at IPL* shows three different methods that execute your code as part of the IPL process.
- *Chapter 6. Adding Your Own Device Support* shows an approach to I/O level programming through the use of EXIO. This chapter shows a technique that can extend the function of a supported device to meet your needs.
- *Chapter 7. Creating Your Own EDL Instruction* explains how to build and add an EDL instruction to the EDL instruction set.
- *Chapter 8. Techniques for Improving Performance* presents several topics and hints to increase performance on your system.

Aids In Using This Book

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

- A Glossary that defines terms and acronyms used in this book and in other EDX library publications
- An Index of topics covered in this book.

A Guide To The Library

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

Contacting IBM About Problems

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

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

Contents

Contents

Contents

Figures

Chapter 1. What is Customization?

The Event Driven Executive consists of a variety of software support you can use in your application. In addition, you can use tools such as utilities to assist you in your operating environment. However, this IBM-supplied software may not provide all the features you require for your application. You can extend or modify the function of several of these facilities to meet your specific operational or application requirements. Extending or modifying these facilities is called *customization*.

This book describes how you can customize some of the EDX software. It also includes a discussion on techniques to improve performance on your Series/1.

Whenever you customize any of the facilities, you should always copy the changes onto a diskette or tape. A subsequent release of EDX or a program temporary fix (PTF) could possibly overlay any customization changes you make to your current release of EDX.

This chapter introduces the facilities you can customize and overviews the performance information presented in this book.

What is Customization?

What You Can Customize

You can customize the following facilities to meet your needs. This book presents some examples of when you might consider customization also.

Operator Commands

You can create your own operator command to perform a function not available with the existing operator commands. For example, you could create an operator command that displays your terminal name and hardware address. On a Series/1 with many terminals attached, this information could be useful.

Chapter 2, "Adding Your Own Operator Command" on page CU-5 contains detailed information on how to create your own operator command.

Session Manager

You can add your application as a new option on an option menu. Further, you can create your own menu screens and procedures to invoke your application.

Chapter 3, "Customizing the Session Manager" on page CU-13 discusses this type of customization.

Task Error Exits

You might consider adding your own task error exit routine to an EDL program. For example, you could do this if the system-supplied routine does not yield all the information you need.

Chapter 4, "Adding Your Own Task Error Exit Routine" on page CU-45 explains how you can perform this type of customization.

Initialization Routines

You can add initialization routines to your system to perform various tasks when you IPL the Series/1. For example, you could have "program A" loaded in partition 1 and the session manager loaded in partition 2. In addition, you could supply a routine to initialize new devices attached to the Series/1.

Chapter 5, "Running Programs and Initialization Routines at IPL" on page CU-55 discusses this type of customization.

What You Can Customize *(continued)*

Device Support

You can extend the system's I/O interface by supplying your own device support. Thus, you can access additional devices not supported under EDX or you can extend the device support EDX does provide.

Chapter 6, "Adding Your Own Device Support" on page CU-63 explains the procedures required to implement such device support.

EDL Instructions

You can create your own Event Driven Language (EDL) instruction to perform operations not available with the existing EDL instruction set.

Chapter 7, "Creating Your Own EDL Instruction" on page CU-83 discusses the details of how to do this.

Improving Performance

You can increase the performance of your system or application in various ways. For example, you can decrease the time it takes the supervisor to access a volume. You can also decrease the compilation time for $EDXASM.

Chapter 8, "Techniques for Improving Performance" on page CU-127 discusses these topics.

Notes

Chapter 2. Adding Your Own Operator Command

If you need a function that is not supported by the existing operator commands, you can create your own routine to perform that function. The Event Driven Executive provides you with an interface that enables you to include your routine in the supervisor. The $U command is reserved for your use. When you add your routine and issue $U, the system invokes the new function.

This chapter explains the steps required to add your own operator command.

Designing and Coding Your Routine

Operator commands run as an ATTNLIST program. Therefore, you must adhere to certain design considerations when you code the routine. A discussion of these design considerations follows.

You must specify MAIN=NO on the PROGRAM statement of your routine.

Code an ENTRY statement specifying $USRCMD following the PROGRAM statement. This statement identifies the entry point to which control is passed when your routine is invoked. Optionally, you can specify a CSECT statement following the PROGRAM statement. The label you specify can be 1–8 characters.

Note: You can omit the ENTRY statement if you use $USRCMD as the label of the CSECT statement.

Designing and Coding Your Routine *(continued)*

Specify the name $USRCMD as the label of your routine. The executable code you provide begins at this label.

You should design your routine so that it executes quickly. Doing this can avoid possible degradation in execution of other tasks. The following instructions are not recommended for use in your routine:

- ENQT/DEQT
- READ/WRITE
- STIMER
- WAIT
- LOAD
- DETACH
- ENDTASK
- TP
- PROGSTOP.

You must code an ENDATTN instruction following the last executable statement in your routine.

Finally, the END statement must be the last statement in your routine.

The source code would look as follows:

```
NEWCMD    PROGRAM     MAIN=NO
          ENTRY       $USRCMD
$USRCMD   EQU         *
          •
          •       (source code for your routine)
          •
          ENDATTN
          END
```

Designing and Coding Your Routine *(continued)*

Some Features You Can Include

You can provide various features in your operator command. The following examples illustrate two features you could provide.

Operator Command for a Specific Terminal

You may want to restrict the function of the operator command to a specific terminal, such as $SYSLOG. By obtaining the terminal name (located in the CCB) from which the command is issued, you could compare the name from the CCB against "$SYSLOG" and branch to an exit upon a "no match" condition.

A cross-partition MOVE, with FKEY=0, is required because the CCB information resides in address space 0 (partition 1).

The following example illustrates how you can obtain and compare terminal names:

```
FETCH     PROGRAM    MAIN=NO
          ENTRY      $USRCMD
          PRINT      OFF
          COPY       CCBEQU           CCB EQUATES
          PRINT      ON
$USRCMD   TCBGET     #1,$TCBCCB       GET ADDR OF CCB
          MOVE       TNAME,($CCBNAME,#1),(8,BYTES),FKEY=0  GET NAME
          IF         (TNAME,NE,SYSLOG,8),GOTO,EXIT         $SYSLOG?
          •
          •      (perform function)
          •
EXIT      ENDATTN
TNAME     TEXT       LENGTH=8
SYSLOG    TEXT      '$SYSLOG',LENGTH=8
          END
```

Designing and Coding Your Routine *(continued)*

Multifunction Operator Command

You might want to have an operator command that provides more than one function. The function executed could depend on the operator input when the command is issued. For example, the operator could enter $U A and the code at label RTNA would be executed. Similarly, if $U B is entered, RTNB is executed; RTNC is executed when you enter $U C. Because no message text is coded on the READTEXT, you must specify A, B, or C when you issue the command.

An example of a how you could develop a multifunction operator command (three routines) follows:

```
MULTI     PROGRAM    MAIN=NO
          ENTRY      $USRCMD
$USRCMD   READTEXT   CMD,PROMPT=COND         GET OPER REQUEST
          IF         (CMD,EQ,C'A',BYTE),GOTO,RTNA
          IF         (CMD,EQ,C'B',BYTE),GOTO,RTNB
          IF         (CMD,EQ,C'C',BYTE),GOTO,RTNC
          GOTO       EXIT                    INVALID REQUEST
RTNA      EQU        *
          •      (perform routine A)
          •
          GOTO       EXIT
RTNB      EQU        *
          •      (perform routine B)
          •
          GOTO       EXIT
RTNC      EQU        *
          •      (perform routine C)
          •
EXIT      ENDATTN
CMD       TEXT       LENGTH=2
          END
```

Testing Your Routine

After you design and code your routine, you should test it. By testing your routine *first* and verifying that it gives you the desired results, you can avoid including an erroneous routine in your supervisor.

You can use the following sample program to verify that your routine meets your requirements:

```
CMDTST   PROGRAM    START
         EXTRN      $USRCMD                  POINTS TO YOUR RTN
         ATTNLIST   (GO,$USRCMD,STOP,STOP)
START    WAIT       ATTNECB,RESET
         PROGSTOP
ATTNECB  ECB
STOP     POST       ATTNECB                  TELL IT WHEN TO QUIT
         ENDATTN
         ENDPROG
         END
```

To test your routine using the sample program, you must do the following:

1. Assemble the sample program (CMDTST) using $EDXASM. The assembled output from this step will be used in step 3.

2. Assemble your routine using $EDXASM. The assembled output from this step will be used in step 3.

3. Link-edit the assembled output from steps 1 and 2 using $EDXLINK. The assembled output from step 1 must be specified on the *first* INCLUDE statement.

4. Upon a successful link-edit (-1 completion code), you can load the program you specified during link-editing.

5. Invoke your routine by pressing the attention key and entering GO. Press the attention key and enter STOP to end the program.

After running the test program, you can determine whether your routine executed as you expected. If the test is successful, you must include your routine in the supervisor.

Including Your Routine in the Supervisor

After a successful test of your new operator command routine, you must link-edit your routine into the supervisor. This section explains how to do this.

Editing Your System INCLUDE Data Set

If you performed a tailored system generation, edit the data set that defines the supervisor modules currently in your supervisor (normally LINKCNTL on EDX002). Otherwise, you must edit $LNKCNTL. Insert the name of the data set and volume containing your routine's assembled output (from step 2 of testing section) *just before* the module EDXINIT. For example, if your assembled output module is named CMDOBJ on volume EDX002, the INCLUDE statement would be as follows:

```
     •
     •
     •
 INCLUDE CMDOBJ,EDX002              YOUR NEW OPERATOR COMMAND
 INCLUDE EDXINIT           *24*     SUPERVISOR INITIALIZATION
 INCLUDE $OVLMGRO          *25*     OVERLAY MANAGER
*INCLUDE RW4963ID          *3*      4963 FIXED HEAD REFRESH SUPPORT
     •
     •
     •
```

After inserting the new INCLUDE statement, save the edited data set in LINKCNTL on EDX002. Next, load $JOBUTIL and specify SUPPREPS when prompted for a data set. SUPPREPS will generate a new supervisor containing your operator command.

Upon completion of the system generation, check the link map listing. The link map will contain the entry and address of $USRCMD if your routine is contained in the supervisor. In addition, if you specified $USRCMD as the label on a CSECT statement, this address will appear also. Initialize your new supervisor (II command of $INITDSK) and IPL the system. You can now invoke your routine using $U as a new operator command.

If $USRCMD appears as an unresolved EXTRN, the ENTRY or CSECT statement specifying $USRCMD was omitted in your routine. You must compile and test the routine again, then perform another system generation.

Operator Command Examples

The following are examples of routines you could use as operator commands:

Message Broadcast Routine

This routine sends a broadcast message to three terminals. The routine is restricted to $SYSLOG. The message text can be up to 60 characters in length. If any of the terminals are in use when the message is sent, the operator is notified. Terminals in use do not receive the broadcast message. You supply the message text when you issue the $U command. For example: "$U SYSTEM IPL IN 5 MINUTES....OPER"

```
BCAST    PROGRAM   MAIN=NO
         ENTRY     $USRCMD
         PRINT     OFF
         COPY      CCBEQU                       CCB EQUATES
         PRINT     ON
$USRCMD  EQU       *
         TCBGET    #1,$TCBCCB                   GET CCB ADDR
         MOVE      TNAME,($CCBNAME,#1),(8,BYTES),FKEY=0  GET NAME
         IF        (TNAME,NE,SYSLOG,8),GOTO,EXIT         $SYSLOG
         READTEXT  MSG,PROMPT=COND,MODE=LINE      READ MESSAGE
         MOVEA     #2,LIST+2                     POINT TO NAMES
         DO         3,TIMES
           MOVE     TNAME,(0,#2),(8,BYTES)      MOVE NAME FROM LIST
           ENQT     TNAME,BUSY=BSYRTN           ENQT TERM
           PRINTEXT MSG                         SEND MESSAGE
           DEQT
           ADD      #2,10                       INCREMENT INDEX
           GOTO     NDU                         BRANCH AROUND BUSY
BSYRTN     EQU       *                          BUSY ROUTINE
           ENQT     $SYSLOG                     NOTIFY OPER. WHICH
           PRINTEXT (0,#2)                      TERMINAL IS BUSY
           PRINTEXT ' IS BUSY'
           DEQT
           ADD      #2,10                       INCREMENT INDEX
NDU      ENDDO
EXIT     ENDATTN
LIST     EQU       *                            LIST OF TERM NAMES
         TEXT      'TERM1',LENGTH=8
         TEXT      'TERM2',LENGTH=8
         TEXT      'TERM3',LENGTH=8
SYSLOG   TEXT      '$SYSLOG',LENGTH=8
MSG      TEXT      LENGTH=60                    MSG HOLD AREA
TNAME    IOCB
         END
```

Operator Command Examples *(continued)*

Display Terminal Name and Address Routine

The following routine displays the terminal name and its address on the terminal from which you issue the command:

```
TERMID    PROGRAM    MAIN=NO
          ENTRY      $USRCMD
          PRINT      OFF
          COPY       CCBEQU                          CCB EQUATES
          PRINT      ON
$USRCMD   EQU        *
          TCBGET     #1,$TCBCCB
          MOVE       TNAME,($CCBNAME,#1),(8,BYTES),FKEY=0       NAME
          MOVE       TADDR+1,($CCBPREP+1,#1),(1,BYTES),FKEY=0   ADDR
          PRINTEXT   '@TERM ID  ADDR@'
          PRINTEXT   TNAME                     PRINT NAME
          PRINTNUM   TADDR,MODE=HEX            PRINT ADDR
          PRINTEXT   '@'
          ENDATTN
TNAME     TEXT       LENGTH=8
TADDR     DATA       F'0'
          END
```

Chapter 3. Customizing the Session Manager

The session manager provides a set of menu screens and procedures that make EDX utilities available for your use. The menu screens enable you to select options (programs) or enter parameters. The procedures invoke the programs you select. By customizing the session manager, you can make a commonly used program a part of a session manager menu. You can do this by modifying existing menus or by creating new menus.

This chapter describes how you customize the session manager. Throughout this chapter, a hypothetical application named PAYROLL is used to show you how to run a program from a newly created menu.

Before you add an application to the session manager, you must ensure the partition in which you load the session manager has enough storage. In addition, you must understand the naming conventions of session manager menus and procedures. You must adhere to these conventions when you add menus and procedures.

How Big Should the Partition Be?

The session manager requires a minimum partition of 16K bytes of storage. When a program, invoked by the session manager, begins execution, the session manager frees 14K bytes of storage. The program you invoke through the session manager must not require more than the partition size minus 2K bytes of storage. For example, if your program requires 34K bytes of storage, the partition must contain at least 36K bytes of available storage.

Customizing the Session Manager

How to Name New Menus and Procedures

Session manager menus and procedures are structured in a hierarchy. The names used for these menus and procedures reflect their level within the hierarchy. Three levels exist:

Primary	Loads programs or presents secondary option menus.
Secondary	Loads programs or presents parameter input menus.
Tertiary	Passes parameters and loads programs.

Menu names must begin with the prefix $SMM. Each menu must have a corresponding procedure. Procedure names must begin with the prefix $SMP.

The menu and procedure names also contain numbers. These numbers are used to indicate the level and option number of the menu. For example, a menu or procedure name containing two numbers indicates the secondary level. Menus or procedures with four numbers indicate the tertiary level.

An example of the naming convention hierarchy follows. The program preparation option along with the $EDXASM option is used:

Primary option menu number	Secondary option menu name	Secondary procedure name	Secondary option menu number	Parm menu name	Procedure ($JOBUTIL) name
Option 2	$SMM02	$SMP02	Option 2	$SMM0202	$SMP0202

Figure 1. Naming convention example

How to Name New Menus and Procedures *(continued)*

Figure 2 illustrates the various paths through which you can invoke programs under the session manager. You can choose any of these paths to invoke programs when you add a new option.

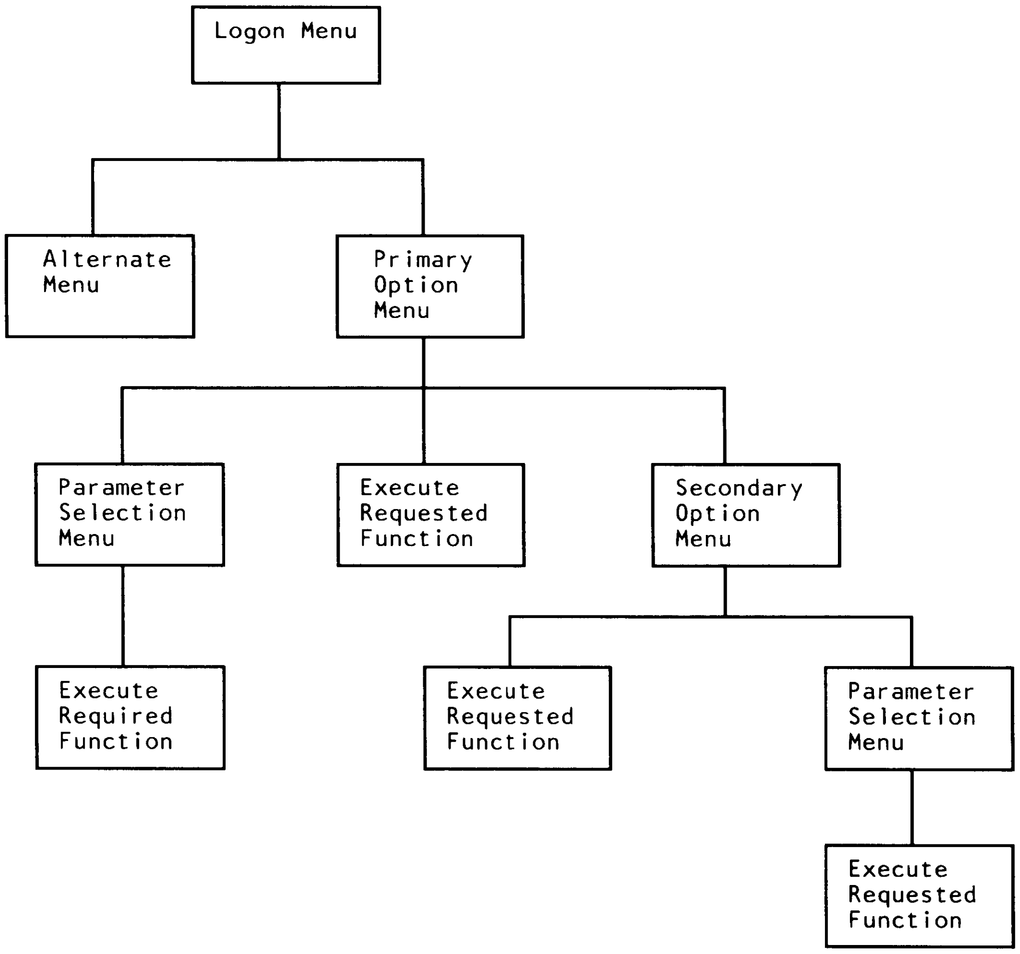


Figure 2. Paths through the session manager

Customizing the Session Manager

Adding an Option to the Primary Option Menu

The primary option menu $SMMPRIM is the first menu presented after you enter your session manager logon ID. You can update this menu to add your program as an option.

This section describes how you can add a program name PAYROLL to the primary option menu. All the steps described are performed using EDX utilities under the session manager.

To add PAYROLL to the primary option menu:

1. Select option 4.4 from the primary option menu. This option loads the $IMAGE utility.

2. Define a null character when the COMMAND(?) prompt appears by entering:

   NULL @

   Note: You may define any character as the null character.

3. Specify the menu to edit when the COMMAND(?) prompt appears by entering:

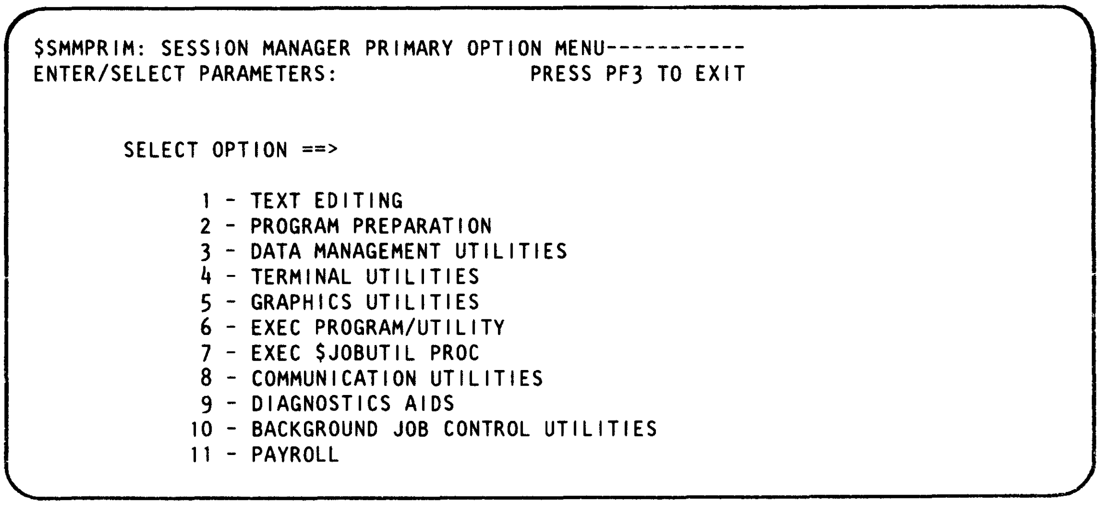
   EDIT $SMMPRIM,EDX002

   The primary option menu $SMMPRIM appears next on the terminal screen.

4. Press the PF1 key to cause the protected fields of menu $SMMPRIM to be displayed as unprotected fields. Doing this enables you to modify the menu. The input data fields are represented by the null character, @, defined in step 2.

5. Position the cursor under the last option number and add the text for the new option, option 11 — PAYROLL.

6. Press the enter key. The enter key takes you out of edit mode. The newly-defined menu image appears as shown in Figure 3.



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


          SELECT OPTION ==>

                 1 - TEXT EDITING
                 2 - PROGRAM PREPARATION
                 3 - DATA MANAGEMENT UTILITIES
                 4 - TERMINAL UTILITIES
                 5 - GRAPHICS UTILITIES
                 6 - EXEC PROGRAM/UTILITY
                 7 - EXEC $JOBUTIL PROC
                 8 - COMMUNICATION UTILITIES
                 9 - DIAGNOSTICS AIDS
                10 - BACKGROUND JOB CONTROL UTILITIES
                11 - PAYROLL
```

Figure 3. Updated session manager primary option menu

7. Press the PF3 key to return to the $IMAGE command mode. In response to the COMMAND(?) prompt, enter:

   SAVE $SMMPRIM,EDX002

8. In response to the message:

   SHOULD THE 3101 DATASTREAM BE SAVED?

   reply N. Reply Y to this message if you use the ATTR command of $IMAGE for a 3101 screen image. Refer to the *Operator Commands and Utilities Reference* for details on the ATTR command of $IMAGE.

At this point, the system saves the updated primary option menu. End the $IMAGE utility (EN command). The primary option menu with PAYROLL is displayed.

Do You Require Additional Menus?

If you are loading a program directly from the primary option menu, you must update the session manager primary procedure. The section "Updating the Primary Procedure" on page CU-33 describes how you can do this.

You can design your new option on the primary option menu so that it consists of several options. To do this, you must create a secondary option menu. The section "Modifying or Creating a Secondary Option Menu" on page CU-18 describes how you can do this.

Adding an Option to the Primary Option Menu *(continued)*

If your program requires input parameters at execution time, you must create a parameter input menu to pass the parameters. The section "Creating a Parameter Input Menu" on page CU-22 describes how you can do this.

Modifying or Creating a Secondary Option Menu

This section describes how you can add a new option to an existing secondary option menu or create your own menu with options. The method you use to add options is similar.

Adding an Option to a Secondary Option Menu

If you want to add your program as an option to a category of programs, you must update an existing secondary option menu.

The following list shows the existing secondary option menus you can update and their categories:

Menu Name	Category
$SMM02	Program preparation
$SMM03	Data management
$SMM04	Terminal utilities
$SMM05	Graphics utilities
$SMM08	Communication utilities
$SMM09	Diagnostic aids
$SMM10	Background Job Control

Figure 4. Existing secondary option menus

Note: All these menus reside on EDX002.

If, for example, you want to add an option that combines both $EDXASM and $UPDATE into one option to the program preparation secondary option menu ($SMM02), you must do the following:

1. Select option 4.4 from the primary option menu. This option loads the $IMAGE utility.

2. Define a null character when the COMMAND(?) prompt appears by entering:

   NULL @

   Note: You may define any character as the null character.

3. Specify the menu to edit when the COMMAND(?) prompt appears by entering:

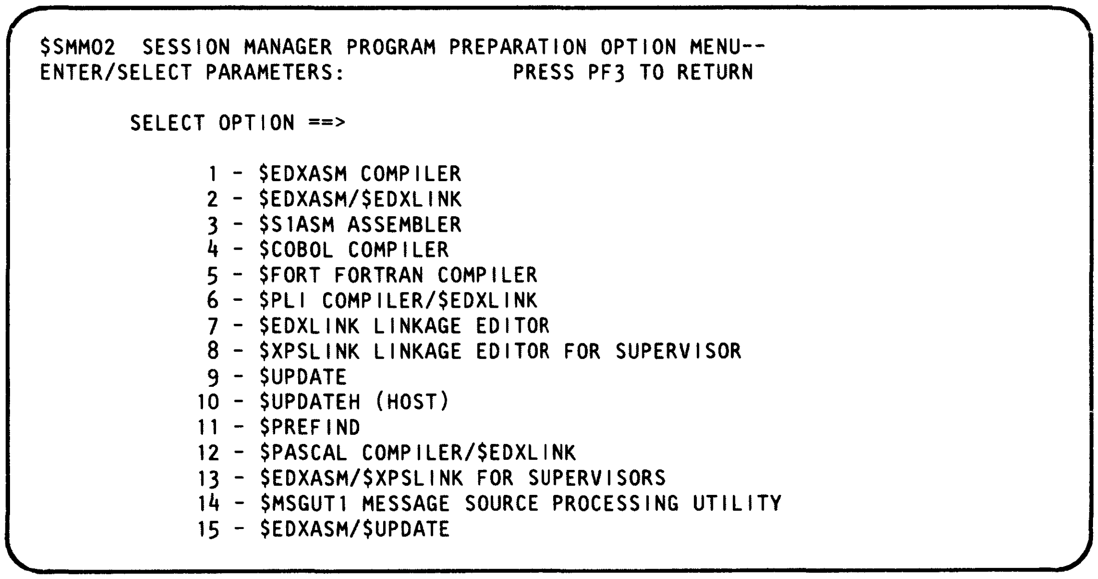
   EDIT $SMM02,EDX002

   The secondary option menu $SMM02 appears next on the terminal screen.

4. Press the PF1 key to cause the protected fields of menu $SMM02 to be displayed as unprotected fields. Doing this enables you to modify the menu. The input data fields are represented by the null character, @, defined in step 2.

5. Position the cursor under the last option number and add the text for the new option, option 15 — $EDXASM/$UPDATE.

6. Press the enter key. The enter key takes you out of edit mode. The newly-defined menu image appears as shown in Figure 5.



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

       SELECT OPTION ==>

              1 - $EDXASM COMPILER
              2 - $EDXASM/$EDXLINK
              3 - $S1ASM ASSEMBLER
              4 - $COBOL COMPILER
              5 - $FORT FORTRAN COMPILER
              6 - $PLI COMPILER/$EDXLINK
              7 - $EDXLINK LINKAGE EDITOR
              8 - $XPSLINK LINKAGE EDITOR FOR SUPERVISOR
              9 - $UPDATE
             10 - $UPDATEH (HOST)
             11 - $PREFIND
             12 - $PASCAL COMPILER/$EDXLINK
             13 - $EDXASM/$XPSLINK FOR SUPERVISORS
             14 - $MSGUT1 MESSAGE SOURCE PROCESSING UTILITY
             15 - $EDXASM/$UPDATE
```

Figure 5. Updated program preparation secondary option menu

7. Press the PF3 key to return to the $IMAGE command mode. In response to the COMMAND(?) prompt, enter:

   SAVE $SMM02,EDX002

Modifying or Creating a Secondary Option Menu *(continued)*

8. In response to the message:

   SHOULD THE 3101 DATASTREAM BE SAVED?

   reply N. Reply Y to this message if you use the ATTR command of $IMAGE for a 3101 screen image. Refer to the *Operator Commands and Utilities Reference* for details on the ATTR command of $IMAGE.

At this point, the system saves the updated secondary option menu. End the $IMAGE utility (EN command).

Creating a Secondary Option Menu

This section describes how you can create a new secondary option menu.

Assume the newly-defined PAYROLL application (option 11 of primary option menu) consists of a mailing list program and a program to print paychecks. To create a menu with these programs as options:

1. Select option 4.4 from the primary option menu. This option loads the $IMAGE utility.
2. Define a null character when the COMMAND(?) prompt appears by entering:

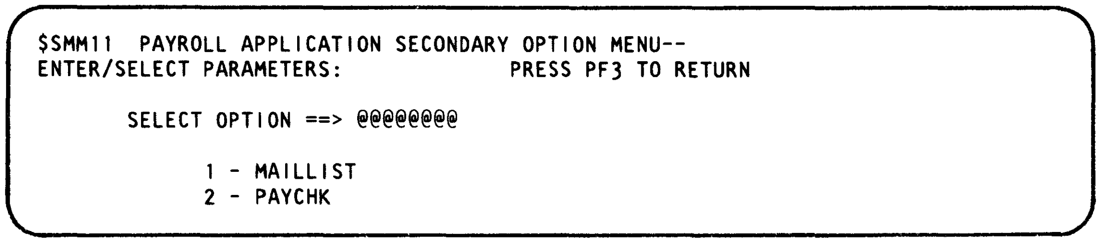
   NULL @

   Note: You may define any character as the null character.
3. Define the screen dimensions as 24 by 80 (full screen) by entering:

   DIMS 24 80
4. Enter the command EDIT. A blank screen appears.
5. Press the PF1 key.

Modifying or Creating a Secondary Option Menu *(continued)*

6. Enter the text for your menu. You must use the null character (defined in step 2) to specify input data fields. Enter eight null characters following the SELECT OPTION text. The secondary option menu for the PAYROLL looks as follows:



```
$SMM11  PAYROLL APPLICATION SECONDARY OPTION MENU--
ENTER/SELECT PARAMETERS:              PRESS PF3 TO RETURN

       SELECT OPTION ==> @@@@@@@@

             1 - MAILLIST
             2 - PAYCHK
```

Figure 6. Sample secondary option menu

7. Press the enter key after you complete the design of your menu. The enter key takes you out of edit mode.

8. Press the PF3 key to return to the $IMAGE command mode.

9. Save your new menu when the COMMAND(?) prompt appears by entering:

   SAVE $SMM10,EDX002

   Note: Use the option number in the name of all related menus. For example, secondary option menu $SMM11 corresponds to option 11 of the primary option menu. Refer to the section "How to Name New Menus and Procedures" on page CU-14 for an explanation of how to name menus.

10. In response to the message:

    SHOULD THE 3101 DATASTREAM BE SAVED?

    reply N. Reply Y to this message if you use the ATTR command of $IMAGE for a 3101 screen image. Refer to the *Operator Commands and Utilities Reference* for details on the ATTR command of $IMAGE.

At this point, the system saves the new secondary option menu. End the $IMAGE utility (EN command).

Do You Require a Parameter Input Menu?

If you are loading a program directly from a secondary option menu, you must update the session manager primary and secondary procedure. The section "Updating the Primary Procedure" on page CU-33 describes how you can do this.

If your program requires input parameters at execution time, you must create a parameter input menu to pass the parameters. The section "Creating a Parameter Input Menu" on page CU-22 describes how you can do this.

Customizing the Session Manager

Creating a Parameter Input Menu

A parameter input menu enables you to pass parameters to the program you want to use. You can use these menus to specify and pass parameters such as data set names, program options, or an output device.

This section shows how to create a parameter input menu for the PAYROLL option and the combined $EDXASM and $UPDATE option.

Assume that the PAYCHK program from the PAYROLL secondary option menu requires three parameters at execution time. The parameters are an input data set, an output data set, and the period end date. To create a menu to pass these parameters:

1. Select option 4.4 from the primary option menu. This option loads the $IMAGE utility.

2. Define a null character when the COMMAND(?) prompt appears by entering:

   NULL @

   Note: You may define any character as the null character.

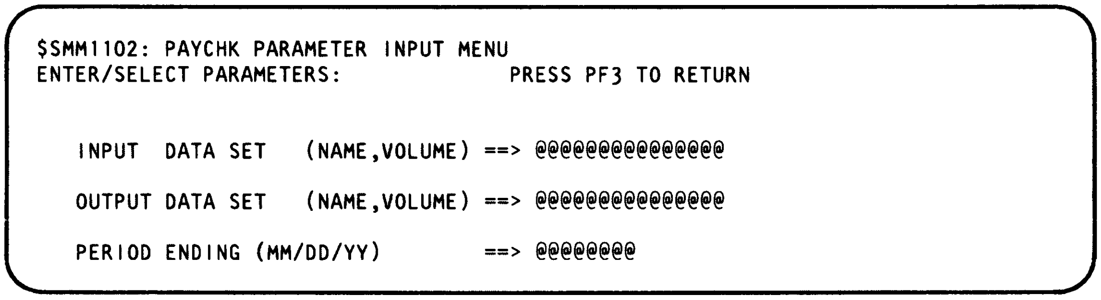
3. Define the screen dimensions as 24 by 80 (full screen) by entering:

   DIMS 24 80

4. Enter the command EDIT. A blank screen appears.

5. Press the PF1 key.

6. Enter the text for your menu. The input data fields are represented by the null character, @, defined in step 2. Note that the menu allows for 15 null characters for the data set and volume name separated by a comma. The parameter input menu for PAYCHK looks as follows:



```
$SMM1102: PAYCHK PARAMETER INPUT MENU
ENTER/SELECT PARAMETERS:                PRESS PF3 TO RETURN


   INPUT  DATA SET   (NAME,VOLUME) ==> @@@@@@@@@@@@@@@

   OUTPUT DATA SET   (NAME,VOLUME) ==> @@@@@@@@@@@@@@@

   PERIOD ENDING (MM/DD/YY)        ==> @@@@@@@@
```

Figure 7. Sample parameter input menu

7. Press the enter key after you complete the design of your menu. The enter key takes you out of edit mode.

8. Press the PF3 key to return to the $IMAGE command mode.

9. Save your new menu by entering:

   SAVE $SMM1102,EDX002

   Note: Use the option number in the name of all related menus. For example, parameter input menu $SMM1102 corresponds to option 2 of the secondary option menu ($SMM11). If your program does not use a secondary option menu, you would name this menu $SMM11. Refer to the section "How to Name New Menus and Procedures" on page CU-14 for an explanation of how to name menus.

10. In response to the message:

    SHOULD THE 3101 DATASTREAM BE SAVED?

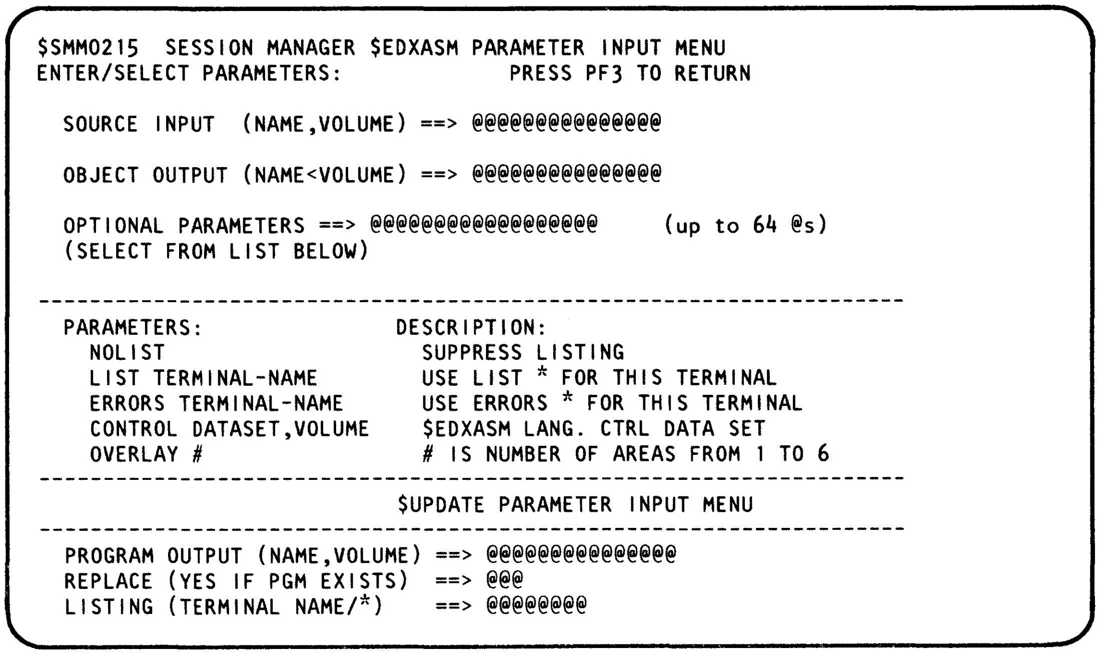
    reply N. Reply Y to this message if you use the ATTR command of $IMAGE for a 3101 screen image. Refer to the *Operator Commands and Utilities Reference* for details on the ATTR command of $IMAGE.

At this point, the system saves the new parameter input menu. End the $IMAGE utility (EN command).

The next step is to write a procedure to pass parameters. See "Writing a Procedure to Pass Parameters" on page CU-25.

Creating a Parameter Input Menu *(continued)*

The same steps are required to create a parameter input menu for the $EDXASM/$UPDATE option discussed in the section "Adding an Option to a Secondary Option Menu" on page CU-18. You can design the menu as shown in Figure 8 . You must save this menu in a data set named $SMM0215.



```
$SMM0215  SESSION MANAGER $EDXASM PARAMETER INPUT MENU
ENTER/SELECT PARAMETERS:            PRESS PF3 TO RETURN

  SOURCE INPUT  (NAME,VOLUME) ==> @@@@@@@@@@@@@@@

  OBJECT OUTPUT (NAME<VOLUME) ==> @@@@@@@@@@@@@@@

  OPTIONAL PARAMETERS ==> @@@@@@@@@@@@@@@@@@     (up to 64 @s)
  (SELECT FROM LIST BELOW)

--------------------------------------------------------------------
  PARAMETERS:               DESCRIPTION:
    NOLIST                    SUPPRESS LISTING
    LIST TERMINAL-NAME        USE LIST * FOR THIS TERMINAL
    ERRORS TERMINAL-NAME      USE ERRORS * FOR THIS TERMINAL
    CONTROL DATASET,VOLUME    $EDXASM LANG. CTRL DATA SET
    OVERLAY #                 # IS NUMBER OF AREAS FROM 1 TO 6
--------------------------------------------------------------------
                            $UPDATE PARAMETER INPUT MENU
--------------------------------------------------------------------
  PROGRAM OUTPUT (NAME,VOLUME) ==> @@@@@@@@@@@@@@@
  REPLACE (YES IF PGM EXISTS)  ==> @@@
  LISTING (TERMINAL NAME/*)    ==> @@@@@@@@
```

Figure 8. $EDXASM and $UPDATE parameter input menu

Writing a Procedure to Pass Parameters

You must write a procedure whenever you pass parameters to your program from a parameter input menu. A procedure consists of two parts:

- PARAMETER section
- $JOBUTIL control statements

To begin writing the procedure:

1. Select option 1, text editing, from the primary option menu. This invokes the $FSEDIT utility.
2. Select option 2 and enter the statements you require for your application.

Writing the PARAMETER Section

The PARAMETER section of the procedure consists of statements unique to the session manager. The PARAMETER statement must be the first statement of your procedure. This section must end with an END statement.

Contained within the PARAMETER section are &PARMnn and &SAVEnn statements. The &PARMnn statements enable your procedure to refer to parameters entered on the menu. The optional &SAVEnn statements save the parameters you enter from session to session.

& PARMnn Statements

To refer to parameters entered on the parameter input menu, you assign a &PARMnn name to each parameter, where nn is the parameter's position number on the menu. You use these names on your $JOBUTIL control statements. Each input field on the menu represents a parameter. For example, MYDS,MYVOL in the field below represents a single parameter and would be assigned the name &PARM01.

```
DATA SET,VOLUME ==> MYDS,MYVOL
```

You assign numbers to parameters in ascending order, from left to right, top to bottom. For example, if a menu contains two parameter entries, you assign the names &PARM01 (first) and &PARM02 (second). The session manager always assigns the name &PARM00 to the 1–4 character session logon ID.

You must end a &PARMnn statement with a period whenever blanks immediately follow that statement.

Writing a Procedure to Pass Parameters *(continued)*

The statements of a procedure that reference two menu entries would look as follows:

```
PARAMETER
&PARM01.
&PARM02.
END
```

The session manager substitutes the &PARMnn names with the actual parameters you enter on the menu. You can use the &PARMnn statements in conjunction with the &SAVEnn statements.

& SAVEnn Statements

The &SAVEnn statements in the procedure enable you to save parameters entered on the menu from session to session. The session manager substitutes &SAVEnn statements with the actual parameters entered on the menu. You can use these statements to save parameters for the menus you create. Once you save a parameter from a menu, the parameter will reappear the next time you access that menu.

The statements of a procedure that reference and save two menu entries would look as follows:

```
PARAMETER
&PARM01,&SAVE01
&PARM02,&SAVE02
END
```

The statement numbers &SAVE61–&SAVE90 are reserved for your use. Use these statement numbers to save parameters from parameter input menus you create.

An example of how to use these statement numbers for the PAYCHK parameter input menu (Figure 7 on page CU-23) follows:

```
PARAMETER
&PARM01,&SAVE61                  (input data set)
&PARM02,&SAVE62                  (output data set)
&PARM03,&SAVE63                  (period end date)
END
```

The menu input fields for EDX utilities have preassigned &SAVE statement numbers (1–60). If you create menus for these utilities and save the input parameters, you must use the preassigned numbers on the &SAVEnn statements. See Figure 9 on page CU-28 for the numbers assigned to the EDX utilities.

Writing a Procedure to Pass Parameters *(continued)*

An example of the statements for the combined $EDXASM/$UPDATE parameter input menu (Figure 8 on page CU-24) follows:

```
PARAMETER
&PARM01,&SAVE01                    (source input)
&PARM02,&SAVE02,&SAVE19            (object output)
&PARM03,&SAVE03                    (compiler options)
&PARM04,&SAVE20                    (pgm name)
&PARM05,&SAVE21                    (replace?)
&PARM06,&SAVE22                    (terminal)
END
```

You can determine which &SAVE statement the session manager assigns to a particular parameter input field by:

1. Using $FSEDIT to list the $SMPxxxx procedure for the utility.

2. Comparing the &PARM and &SAVE statements from the listing with the parameter input menu the session manager uses for that utility.

The procedure you write must pass parameters to each utility in the order shown in the $SMPxxxx procedure.

Writing a Procedure to Pass Parameters *(continued)*

The following figure shows the preassigned numbers for the EDX utilities:

Statement	Procedure	Utility / Function
&SAVE01-03	$SMP0201	$EDXASM
&SAVE04-06	$SMP0203	$S1ASM
&SAVE07-13	$SMP0204	$COBOL
&SAVE14-16	$SMP0205	$FORT
&SAVE17-18	$SMP0208	$EDXLINK, $XPSLINK
&SAVE19-22	$SMP0209	$UPDATE
&SAVE23-24	$SMP0211	$PREFIND
&SAVE25-26	$SMP0308	$MOVEVOL
&SAVE27	$SMP0405	$FONT
&SAVE28	$SMP0501	$DIUTIL
&SAVE29	$SMP0502	$DICOMP
&SAVE30	$SMP0503	$DIINTR
&SAVE31-35	$SMP06	Execute application program
&SAVE36	$SMP0801	$BSCTRCE
&SAVE37	$SMP0806	$PRT2780
&SAVE38	$SMP0807	$PRT3780
&SAVE39	$SMP0808	$HCFUT1
&SAVE40-41	$SMP0211	$PREFIND
&SAVE42	$SMP0207	$EDXLINK
&SAVE43	$SMP0901	$DUMP
&SAVE44	$SMP0208	$XSPLINK
&SAVE45-49	$SMP0206	$PLI
&SAVE45-50	$SMP0212	$PASCAL
&SAVE51	$SMP8101	$ARJE
&SAVE52-58	$SMP0904	$VERIFY
&SAVE59	$SMP0204	$COBOL
&SAVE60	Reserved	

Figure 9. &SAVEnn numbers of EDX utilities/functions

Writing a Procedure to Pass Parameters *(continued)*

Writing the $JOBUTIL Control Statements

The procedure you write must use $JOBUTIL control statements. The session manager passes the statements in this part of the procedure to $JOBUTIL, which then loads and executes the program. The $JOBUTIL control statements are described in detail in the *Operator Commands and Utilities Reference.*

This section shows examples of $JOBUTIL control statements used in conjunction with &PARMnn statements. Use the examples presented as a guide as you write your procedure.

Three examples are shown. The first example is the procedure required to invoke $EDXASM. The remaining examples show the procedures for the new options, PAYCHK and $EDXASM/$UPDATE.

You must enter $JOBUTIL control statements in the following format:

Command	Position 1 to 8
Operand	Position 10 to 17
Comment	Position 18 to 71

Saving the Procedure

After you enter the statements, do the following:

1. Return to the $FSEDIT primary option menu by entering MENU on the command line.

2. Select option 4 and specify the data set name in which the new procedure is to be saved. Specify EDX002 as the volume name.

   Procedure names can be a maximum of eight characters in length ($SMPxxxx) and must have the prefix $SMP. The "xxxx" portion of the name should contain the numbers that reflect the option number on the primary option menu and the option number on the secondary option menu (if you use one).

   However, procedure names **must** correspond with the name of the parameter input menu. For example, you name the procedure for the PAYCHK program $SMP1102. This name corresponds to the name of the parameter input menu $SMM1102. Similarly, you name the procedure for the $EDXASM/$UPDATE option $SMP0215. This name corresponds to the parameter input menu $SMM0215. Refer to the section "How to Name New Menus and Procedures" on page CU-14 for an explanation of how to name procedures.

Writing a Procedure to Pass Parameters *(continued)*

3. After you save the procedure, enter option 8 to exit $FSEDIT and return to the session manager.

The next step is updating the session manager's primary and/or secondary procedure. The section "Updating the Primary Procedure" on page CU-33 explains how you can do this.

Examples of Procedures

Use the examples shown in this section as a guide for the procedures you write.

The session manager uses many different procedure formats. You can write more sophisticated procedures by copying existing session manager procedures and updating them with the $FSEDIT utility to invoke different programs and save parameters.

$EDXASM Procedure

```
PARAMETER
&PARM01,&SAVE01              (source input)
&PARM02,&SAVE02              (object output)
&PARM03,&SAVE03              (compiler options)
END
LOG        OFF
REMARK     @ASSEMBLE &PARM01. TO &PARM02. USERID=&PARM00.
JOB        $SMP0201
PROGRAM    $EDXASM,ASMLIB
PARM       &PARM03.
DS         &PARM01.
DS         $SM1&PARM00.,EDX003     (work data set)
DS         &PARM02.
EXEC
EOJ
END
```

Figure 10. Procedure to invoke $EDXASM

Writing a Procedure to Pass Parameters *(continued)*

PAYCHK Procedure

Note that the parameters passed are saved in &SAVE61–&SAVE63. The parameter input menu for this procedure is shown in Figure 7 on page CU-23.

```
PARAMETER
&PARM01,&SAVE61              (input data set)
&PARM02,&SAVE62              (output data set)
&PARM03,&SAVE63              (period end date)
END
LOG        OFF
REMARK    @PAYROLL PAYCHECK PROCEDURE   USERID=&PARM00.
JOB        $SMP1102
PROGRAM    PAYCHK,MYVOL
PARM       &PARM03.
DS         &PARM01.
DS         &PARM02.
EXEC
EOJ
END
```

Figure 11. Procedure to invoke PAYCHK

Writing a Procedure to Pass Parameters *(continued)*

$EDXASM/$UPDATE Procedure

This procedure combines the session manager procedure for $EDXASM and $UPDATE into one procedure. Note that &PARM02 is saved to &SAVE02 and &SAVE19 in one statement. The parameter input menu for this procedure is shown in Figure 8 on page CU-24.

```
PARAMETER
&PARM01,&SAVE01                    (source input)
&PARM02,&SAVE02,&SAVE19            (object output)
&PARM03,&SAVE03                    (compiler options)
&PARM04,&SAVE20                    (pgm name)
&PARM05,&SAVE21                    (replace?)
&PARM06,&SAVE22                    (terminal)
END
LOG       OFF
REMARK   @ASSEMBLE &PARM01. TO &PARM02. USERID=&PARM00.
JOB       $SMP0215
PROGRAM   $EDXASM,ASMLIB
PARM      &PARM03.
DS        &PARM01.
DS        $SM1&PARM00.,EDX003          (work data set)
DS        &PARM02.
EXEC
JUMP      EXIT,NE,-1
REMARK   @CREATE LOAD MODULE &PARM02. TO &PARM04.
PROGRAM   $UPDATE,EDX002
PARM      &PARM06. &PARM02. &PARM04. &PARM05.
EXEC
LABEL     EXIT
EOJ
END
```

Figure 12. Procedure to invoke $EDXASM/$UPDATE

Updating the Primary Procedure

You must update the session manager primary procedure ($SMPPRIM) whenever you add an option to the primary option menu or to a secondary option menu. The primary procedure contains all option numbers as well as menu and program names associated with all options.

This section explains how you can update the primary procedure for options you add.

Perform the following steps to update the primary procedure ($SMPPRIM) for a new option:

1. Select option 1 (text editing) on the primary option menu and press the enter key. The next menu to appear on the terminal screen is the primary option menu for $FSEDIT.

2. Select option 3 (read) and specify **$SMPPRIM** as the data set name. Specify **EDX002** as the volume name. Press the enter key.

3. After the utility reads $SMPPRIM into your work data set, enter option 2 (edit) to update $SMPPRIM.

Entering Changes to the Primary Procedure

The option number you specify can be either a number or a letter. Follow the format of $SMPPRIM as you enter option numbers, program, and menu names.

Program with No Parameters

Assume the new option (11 — PAYROLL) on the primary option menu is a program that does not require parameters (can be loaded directly). To update $SMPPRIM, scroll to the bottom (PF3 key) and add the new option number and program name. You would update $SMPPRIM to look like the following:

```
           •
           •
           •
'9       ',$SMM09            DIAGNOSTICS SECONDARY OPTION MENU
'9.1     ',$SMM0901          $DUMP PARM INPUT MENU
'9.2     ',*$DISKUT2EDX002   EXECUTE $DISKUT2
'9.3     ',*$IOTEST EDX002   EXECUTE $IOTEST
'9.4     ',$SMM0904          $VERIFY PARM INPUT MENU
'10      ',$SMM10            $JOBQUT/$SUBMIT OPTION MENU
'10.1    ',*$JOBQUT EDX002   EXECUTE $JOBQUT
'10.2    ',*$SUBMIT EDX002   EXECUTE $SUBMIT
'11      ',*PAYROLL EDX002   EXECUTE PAYROLL PROGRAM
```

Figure 13. Example of a program added with no parameters

The asterisk before the program name indicates the program does not require parameters when loaded.

Updating the Primary Procedure *(continued)*

Optionally, you could pass a data set and volume name to a program. You might want to do this if your program normally prompts you for a data set after you load the program. For example, $FSEDIT and $EDXLINK prompt you for a work data set when you load them. You can pass your program one of the session manager work data sets or a data set you create. An asterisk must precede and follow the data set name (padded to eight characters in length).

The following example shows how to use a session manager work data set:

```
'1    ',*$FSEDIT EDX002*$SME&    *EDX003
```

$FSEDIT uses the session manager work data set $SMEuser, where "user" is your 1–4 character logon ID.

If you append an **&** to the data set name $SME, the session manager replaces the **&** with your logon ID.

The next example shows how to pass a program the data set WORKDS1 on volume MYVOL:

```
'1    ',*$FSEDIT EDX002*WORKDS1 *MYVOL
```

At this point, you must save $SMPPRIM. Refer to the section "Saving the Primary Procedure" on page CU-37 for information on saving $SMPPRIM. After you save $SMPPRIM, you can invoke your new option from the primary option menu.

Updating the Primary Procedure *(continued)*

Program Using Parameter Input Menu Only

If the new option (11 — PAYROLL) required only a parameter input menu, you would update $SMPPRIM as shown in Figure 14. To update $SMPPRIM in this case, scroll to the bottom (PF3 key) and add the new option number and the name of the parameter input menu.

Note: The session manager searches for a procedure on EDX002 that corresponds to the name of the parameter input menu. For example, to load the program for $SMM1102, the session manager would search EDX002 for a procedure named $SMP1102.

```
          •
          •
          •
'9      ',$SMM09            DIAGNOSTICS SECONDARY OPTION MENU
'9.1    ',$SMM0901          $DUMP PARM INPUT MENU
'9.2    ',*$DISKUT2EDX002   EXECUTE $DISKUT2
'9.3    ',*$IOTEST EDX002   EXECUTE $IOTEST
'9.4    ',$SMM0904          $VERIFY PARM INPUT MENU
'10     ',$SMM10            $JOBQUT/$SUBMIT OPTION MENU
'10.1   ',*$JOBQUT EDX002   EXECUTE $JOBQUT
'10.2   ',*$SUBMIT EDX002   EXECUTE $SUBMIT
'11     ',$SMM1102          EXECUTE PAYROLL PROGRAM
```

Figure 14. Example of a program added with parameter input menu

After you make the entry, you must save $SMPPRIM. Refer to the section "Saving the Primary Procedure" on page CU-37 for information on saving $SMPPRIM. After you save $SMPPRIM, you can invoke your new option from the primary option menu.

Updating the Primary Procedure *(continued)*

Program Using Secondary Option Menu

The PAYROLL example shown throughout this chapter is a new option on the primary option menu but also uses a secondary option menu. To update $SMPPRIM, scroll to the bottom (PF3 key) and make the entries as shown in Figure 15. An explanation of the entries follows the figure.

```
               •
               •
               •
'9        ',$SMM09          DIAGNOSTICS SECONDARY OPTION MENU
'9.1      ',$SMM0901        $DUMP PARM INPUT MENU
'9.2      ',*$DISKUT2EDX002 EXECUTE $DISKUT2
'9.3      ',*$IOTEST EDX002 EXECUTE $IOTEST
'9.4      ',$SMM0904        $VERIFY PARM INPUT MENU
'10       ',$SMM10          $JOBQUT/$SUBMIT OPTION MENU
'10.1     ',*$JOBQUT EDX002 EXECUTE $JOBQUT
'10.2     ',*$SUBMIT EDX002 EXECUTE $SUBMIT
'11       ',$SMM11          PAYROLL SECONDARY OPTION MENU
'11.1     ',*MAILLISTMYVOL  EXECUTE MAILING LIST PROGRAM
'11.2     ',$SMM1102        PAYCHECK PARM INPUT MENU
```

Figure 15. Example of program added using secondary option menu

The entry for option 11 points to the secondary option menu $SMM11 (Figure 6 on page CU-21). The entry for option 11.1 points to the program MAILLIST on volume MYVOL. MAILLIST requires no parameters when the session manager loads it. The entry for option 11.2 points to the parameter input menu $SMM1102 (Figure 7 on page CU-23) for the PAYCHK program.

Note: The session manager searches for a procedure on EDX002 that corresponds to the name of the parameter input menu. For example, to load the program for $SMM1102, the session manager would search EDX002 for a procedure named $SMP1102.

You would perform similar update steps to add the $EDXASM/$UPDATE example discussed in "Adding an Option to a Secondary Option Menu" on page CU-18. For this example, you enter option number 2.15 and the menu name $SMM0215 as shown in Figure 16 .

```
            •
            •
            •
'2.11    ',$SMM0211        $PREFIND PARM INPUT MENU
'2.12    ',$SMM0212        $PASCAL/$EDXLINK PARM INPUT MENU
'2.13    ',$SMM0213        $EDXASM/$XPSLINK PARM INPUT MENU
'2.14    ',*$MSGUT1 EDX002*$SM1&   *EDX003
'2.15    ',$SMM0215        NEW $EDXASM/$UPDATE OPTION
```

Figure 16. Example of adding $EDXASM/$UPDATE option

At this point, you must save $SMPPRIM. Refer to the section "Saving the Primary Procedure" for information on saving $SMPPRIM. After you save $SMPPRIM, you must update or create a secondary procedure. The section "Updating or Creating a Secondary Procedure" on page CU-38 explains how to do this.

Saving the Primary Procedure

When you complete the updating of $SMPPRIM, do the following:

1. Enter MENU in the command field to return to the $FSEDIT menu.

2. Select option 4 from the $FSEDIT primary option menu. Respond YES to the prompt message to write the updated procedure back to $SMPPRIM on volume EDX002.

3. Enter option 8 to end $FSEDIT and return to the session manager primary option menu.

Customizing the Session Manager

Updating or Creating a Secondary Procedure

You must update a secondary procedure whenever you add an option to an existing secondary option menu. Further, if you create a new secondary option menu you must create a secondary procedure for that option menu.

The format of a secondary procedure is almost identical to the format of the primary procedure ($SMPPRIM). A secondary procedure contains option numbers and menu and program names that pertain only to a specific secondary option menu.

All secondary procedures begin with the name $SMPxx, where xx is the number from the primary option menu. For example, $SMP04 is the secondary procedure for terminal utilities (option 4).

Updating an Existing Secondary Procedure

To show you how to add an option to an existing secondary procedure ($SMP02), the $EDXASM/$UPDATE example (Figure 5 on page CU-19) is used.

Perform the following steps to update $SMP02:

1. Select option 1 (text editing) on the primary option menu and press the enter key. The next menu to appear on the terminal screen is the primary option menu for $FSEDIT.

2. Select option 3 (read) and and specify $SMP02 as the data set name. Specify EDX002 as the volume name.

3. After the utility reads $SMP02 into your work data set, enter option 2 (edit) to update $SMP02.

4. Scroll to the bottom (PF3 key) and enter the new option number and the name of the parameter input menu (Figure 8 on page CU-24).

The following is an example of the updated $SMP02 procedure:

```
 SELECTION $SMP02
'1       ',$SMM0201        $EDXASM PARM INPUT MENU
'2       ',$SMM0202        $EDXASM/$EDXLINK PARM INPUT MENU
'3       ',$SMM0203        $S1ASM PARM INPUT MENU
'4       ',$SMM0204        $COBOL PARM INPUT MENU
'5       ',$SMM0205        $FORT PARM INPUT MENU
'6       ',$SMM0206        $PLI/$EDXLINK PARM INPUT MENU
'7       ',$SMM0207        $EDXLINK PARM INPUT MENU
'8       ',$SMM0208        $XPSLINK FOR SUPERVISORS PARM INPUT MENU
'9       ',$SMM0209        $UPDATE PARM INPUT MENU
'10      ',*$UPDATEHEDX002 EXECUTE $UPDATEH
'11      ',$SMM0211        $PREFIND PARM INPUT MENU
'12      ',$SMM0212        $PASCAL/$EDXLINK PARM INPUT MENU
'13      ',$SMM0213        $EDXASM/$XPSLINK PARM INPUT MENU
'14      ',*$MSGUT1 EDX002*$SM1&   *EDX003
'15      ',$SMM0215        NEW $EDXASM/$UPDATE OPTION
END
```

Figure 17. Updated $SMP02 secondary procedure

Saving an Existing Secondary Procedure

When you complete the updating of $SMP02, do the following:

1. Enter MENU in the command field to return to the $FSEDIT menu.

2. Select option 4 from the $FSEDIT primary option menu. Respond YES to the prompt to write the updated procedure back to $SMP02 on volume EDX002.

3. Enter option 8 to end $FSEDIT and return to the session manager primary option menu.

After completing these steps, you can invoke the new option from either the primary or secondary option menu.

Updating or Creating a Secondary Procedure *(continued)*

Creating a Secondary Procedure

To show you how to create a new secondary procedure, the PAYROLL example (Figure 6 on page CU-21) is used.

A simple way to create a new secondary procedure is to edit an existing secondary procedure. You can add the appropriate entries you need for your program and delete the entries you do not need. By editing an existing secondary procedure, you can ensure that the required format remains correct. All existing secondary procedures are named $SMPxx, where xx is an option number.

Perform the following steps to create a new secondary procedure:

1. Select option 1 (text editing) on the primary option menu and press the enter key. The next menu to appear on the terminal screen is the primary option menu for $FSEDIT.

2. Select option 3 (read) and specify the data set name of an existing secondary procedure, for example $SMP02. Specify EDX002 as the volume name.

3. After the utility reads $SMP02 into your edit work data set, enter option 2 (edit) to edit $SMP02.

4. Keeping the same format, replace the entries in $SMP02 with the entries for PAYROLL.

The following is an example of the secondary procedure for PAYROLL:

```
 SELECTION $SMP11
'1       ',*MAILLIST        MAILING LIST PROGRAM
'2       ',$SMM1102         PAYCHK PARM INPUT MENU
END
```

Figure 18. New secondary procedure for PAYROLL

Saving a New Secondary Procedure

When you complete the updating, do the following:

1. Enter MENU in the command field to return to the $FSEDIT menu.

2. Select option 4 from the $FSEDIT primary option menu. Specify the *new* data set name which will contain the secondary procedure. For this example, enter $SMP11 as the new data set name. $FSEDIT will create this data set for you. Specify EDX002 as the volume name. Respond YES to the prompt message after you specify the new data set name.

3. Enter option 8 to end $FSEDIT and return to the session manager primary option menu.

After completing these steps, you can invoke the new option from the primary option menu.

Using an Alternate Session Menu

When you log on to the session manager, you can override the menu presentation by specifying an option menu that you have created. You might consider this method to provide menus tailored to your system.

You can use the ALTERNATE SESSION MENU prompt below the user ID prompt if you create your own menus and procedures. Entering the name of your menu as an alternate causes your menu to appear instead of the session manager primary option menu.

When you use this method of customizing the session manager:

1. Adhere to the naming conventions discussed in the section "How to Name New Menus and Procedures" on page CU-14.

2. Ensure the menus and associated procedures reside on volume EDX002.

3. Design the menus and procedures as discussed throughout this chapter.

The following example shows the logon menu with the name of an alternate menu, $SM9901, specified:

```
$SMMLOG: THIS TERMINAL IS LOGGED ON TO THE SESSION MANAGER
                                                   17:55:31
ENTER 1-4 CHAR USER ID ==> MYID                   12/11/83
(ENTER LOGOFF TO EXIT)

ALTERNATE SESSION MENU ==> $SM9901
(OPTIONAL)
```

Figure 19. Session manager logon screen with alternate menu

Customizing the Session Manager

How to Modify Data Set Allocation and Deletion

The session manager allocates and deletes temporary data sets when you logon and logoff respectively. The session manager uses these data sets as work data sets for the various programs it invokes. Two session manager data sets control allocation and deletion. $SMALLOC controls the data sets to be allocated. $SMDELET controls the data sets to be deleted.

You can tailor the work data set allocations and deletions by modifying the $SMALLOC and $SMDELET data sets with $FSEDIT or $EDIT1N. Modifications usually consist of changing the size or volume name of a data set. However, you can also allocate and delete up to four additional data sets.

You can use these additional temporary data sets for programs you use. For example, if your program needed to write data to a temporary data set then later retrieve data from that data set, you could run your program under the session manager and have the session manager create that data set.

Figure 20 lists all the session manager data sets with sizes and functions.

Data set name	Size in 256 EDX records	Function
$SMEuser	400	Used by $FSEDIT as a work data set.
$SMPuser	30	Used by session manager to save input parameters from session to session. This data set is not deleted at logoff.
$SMWuser	30	Used by session manager to submit procedures to $JOBUTIL.
$SM1user	400 [1]	Used by $S1ASM, $EDXASM, $COBOL, $PASCAL, $PLI, and $FORT as a work data set.
$SM2user	400 [1]	Used by $EDXLINK, $S1ASM, $EDXASM, $COBOL, $PLI, and $FORT as a work data set.
$SM3user	250 [1]	Used by $S1ASM, $COBOL, $PASCAL, and $PLI as a work data set.

Figure 20. Data sets created by the session manager

Note: The session manager substitutes your logon ID for "user" and appends your logon ID to the data set name.

[1] Using the assemblers and compilers noted may require that you delete and reallocate these data sets to a larger size. Recommended sizes are 2000 for $SM1 and $SM2, and 800 for $SM3.

How to Modify Data Set Allocation and Deletion *(continued)*

Allocating Data Sets

In addition to allocating data sets $SM1 through $SM3, you can allocate data sets $SM4 through $SM7. The default size of these data sets is 100 records.

The following is an example of how $SMALLOC looks:

```
$SMP      00        EDX003    NAME AND VOLME FOR OPEN
$SMP      30        EDX003    SIZE AND VOLUME TO ALLOCATE
$SMW      30        EDX003    SIZE AND VOLUME TO ALLOCATE
$SME      400       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM1      400       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM2      400       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM3      400       EDX003    SIZE AND VOLUME TO ALLOCATE
END       *** TERMINATOR - INDICATES END OF ALLOCATED DATASETS ***
$SM4      100       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM5      100       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM6      100       EDX003    SIZE AND VOLUME TO ALLOCATE
$SM7      100       EDX003    SIZE AND VOLUME TO ALLOCATE
*****************************************************************
*****************************************************************
**   $SMLOG WORK DATASET PARAMETER VALUES FOR ALLOCATE FUNCTION  **
**        NOTE: THE DATASETS $SMW AND $SMP MUST RESIDE ON        **
**              THE VOLUME EDX003.  ALL OTHERS MAY BE REASSIGNED **
**        NOTE: THE FIRST ENTRY IN THIS LIST IS USED TO TEST FOR **
**              THE EXISTENCE OF THE $SMP DATASET. DON'T DELETE. **
**              5719-UT5 COPYRIGHT IBM CORP 1980                 **
*****************************************************************
*****************************************************************
          END
*****************************************************************
```

Figure 21. $SMALLOC data set

If you want $SM4 allocated, move the END statement (in column 1) to follow $SM4. The END statement indicates the end of the list of data sets to be allocated. If you add data sets to the list in $SMALLOC, you should also add names of the data sets to $SMDELET. If you change the volume name of a work data set in the $SMALLOC and $SMDELET data sets, then you have to change all the session manager procedures that use that work data set. After you complete your modifications, you must save the updated $SMALLOC data set.

The only required data sets are $SMP and $SMW. You must allocate these data sets on volume EDX003.

How to Modify Data Set Allocation and Deletion *(continued)*

Deleting Data Sets

Before you end the session manager, the session manager prompts you for the disposition of the data sets. The data sets to be deleted are normally the data sets that were allocated at the start of the session. Enter a Y to save the data sets or an N to delete the data sets.

Note: Abnormal termination of the session manager prevents the deletion of the temporary data sets.

If you add data set names in $SMALLOC, you must also update $SMDELET with those data set names. Update $SMDELET in a similar manner to $SMALLOC. The END statement (in column 1) indicates the last data set to be deleted. After you complete your modifications, you must save the updated $SMDELET data set.

Figure 22 lists the contents of $SMDELET.

```
$SME            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM1            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM2            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM3            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SMW            EDX003    PREFIX NAME AND VOLUME TO DELETE
END      *** TERMINATOR - INDICATES END OF DATA SETS TO BE DELETED ***
$SM4            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM5            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM6            EDX003    PREFIX NAME AND VOLUME TO DELETE
$SM7            EDX003    PREFIX NAME AND VOLUME TO DELETE
***************************************************************
***************************************************************
**  $SMEND WORK DATASET PARAMETER VALUES FOR DELETE FUNCTION **
**             5719-UT5 COPYRIGHT IBM CORP 1980              **
***************************************************************
***************************************************************
         END
***************************************************************
```

Figure 22. $SMDELET data set

Chapter 4. Adding Your Own Task Error Exit Routine

When a program is executing, an exception condition may occur either in the program itself or in the Series/1 processor. If an exception occurs, the supervisor invokes its error handling routine, displays diagnostic information in the form of a program check message on $SYSLOG, and cancels the program. You can provide your own exception handling routine by writing a task error exit routine.

When you provide a task error exit routine in your program, the supervisor passes control to your EDL routine when an exception occurs. Your routine can then capture and format status information specific to your program.

Some of the processing your task error exit routine could perform is:

- Releasing any enqueued resources such as event control blocks (ECBs) or queue control blocks (QCBs).
- Displaying, on all terminals currently being used by the program, a message that would inform the operator(s) of a malfunction and the appropriate action to be taken.
- Printing the data set control blocks (DSCBs) from the program header and the program.
- Printing the input/output control blocks (IOCBs), terminal control blocks (CCBs), and task control blocks (TCBs) in your application.
- Printing any sensor based I/O control blocks (SBIOCBs) or any other data special to your application.
- Reloading your program or loading another program.

Adding Your Own Task Error Exit Routine

You can:

- Extend the system-supplied task error exit routine ($$EDXIT).
- Provide your own routine independent of $$EDXIT.

You specify the EDL entry point name of the task error exit routine on the ERRXIT= operand of the PROGRAM or TASK statement.

This chapter describes how to extend the system-supplied task error exit routine or create your own task error exit routine.

Extending the System-Supplied Task Error Exit Routine

The system-supplied task error exit routine ($$EDXIT) prints and displays general information regarding an exception check. An example of the output you get is shown in Figure 23. The *Problem Determination Guide* discusses this exception output in detail.

```
            ******************************************
            * WARNING!!  AN EXCEPTION HAS OCCURRED!! *
            ******************************************

PROGRAM NAME              = PCHECK       PSW = 8002
PROGRAM VOLUME            = EDXWRK       IAR = 2AD6
PROGRAM LOAD POINT        =   0000       AKR = 0110
ADDRESS OF ACTIVE TCB     =   0120       LSR = 80D0
ADDRESS OF CCB            =   0F5E       R0 (WORK REGISTER)   = 0064
NUMBER OF DATA SETS       =      1       R1 (EDL INSTR ADDR)  = 010A
NUMBER OF OVERLAYS        =      0       R2 (EDL TCB ADDR)    = 0120
$TCBADS                   =   0001       R3 (EDL OP1 ADDR)    = 0037
ADDRESS OF FAILURE                       R4 (EDL OP2 ADDR)    = 0034
  (REL.TO PGM LOAD POINT) =   010A       R5 (EDL COMMAND)     = 015C
DUMP OF FAIL ADDRESS                     R6 (WORK REGISTER)   = 0000
 010A: 015C 0000 0034 8332               R7 (WORK REGISTER)   = 0000
$TCBCO  =      -1 DEC;  FFFF HEX         #1 =  0037
$TCBCO2 =       0 DEC;  0000 HEX         #2 =  0000

PSW ANALYSIS:

  SPECIFICATION CHECK
  TRANSLATOR ENABLED
```

Figure 23. Sample output from $$EDXIT

How to Code the Task Error Exit Extension

$$EDXIT contains a WXTRN statement for a routine called PCHKRTN. If PCHKRTN exists, $$EDXIT passes control to PCHKRTN after printing the exception check data on $SYSPRTR. Use PCHKRTN as the extension to $$EDXIT.

To provide your routine as an extension to $$EDXIT, you must:

- Specify MAIN=NO on the PROGRAM statement of your routine.
- Code an ENTRY statement specifying PCHKRTN.
- Specify PCHKRTN as the label of your routine. The executable code you provide begins at this label.
- Specify a PROGSTOP statement following the executable code.
- Specify the END statement as the last statement of your routine.

For example:

```
ERRRTN    PROGRAM   MAIN=NO
          ENTRY     PCHKRTN
PCHKRTN   EQU       *
          •
          •        (source code for your routine)
          •
          PROGSTOP
          END
```

Link-Editing the Task Error Exit Extension

After you assemble your routine, link-edit the assembled output with your main program and $$EDXIT. The system includes $$EDXIT in the link-edit when you specify an AUTOCALL statement referencing $AUTO,ASMLIB. The following is an example of the link control statements you pass to $EDXLINK.

```
INCLUDE     MAINOBJ,MYVOL               (includes main pgm)
AUTOCALL    $AUTO,ASMLIB                (includes $$EDXIT)
INCLUDE     PCHKOBJ,MYVOL               (includes your routine)
LINK        MAINPGM,MYVOL REPLACE END
```

Adding Your Own Task Error Exit Routine

Creating Your Own Task Error Exit Routine

This section explains how you can create your task error exit routine. A sample program is also shown to assist you in coding the routine.

Defining the Task Error Exit Control Block

When you create your own task error exit routine, you must define an area of storage called a task error exit control block (TEECB). The TEECB provides the linkage between the supervisor and your routine. The supervisor stores hardware status information in the TEECB when an exception occurs. You must define the TEECB area even if your routine does not use the status information.

You must align the TEECB on a fullword boundary. The TEECB has the following format:

```
         ALIGN     WORD            ALIGN ON FULLWORD BOUNDARY
TEECB    EQU       *
TEECTL   DC        X'0002'         CONTROL WORD
TEESIA   DC        A(EXITRTN)      ADDRESS OF STARTING INSTRUCTION
TEEHSA   DC        A(HSA)          ADDRESS OF HARDWARE STATUS AREA
```

Figure 24. Format of the task error exit control block (TEECB)

In the first word (TEECTL), bits 0–7 are reserved and must be zero. Bits 8–15 specify the number of data words that follow. Always code X'0002' as the value of this word.

The second word (TEESIA) contains the starting instruction address (SIA) of your task error exit routine.

The last word (TEEHSA) contains the address of a storage area you reserve to receive the hardware status information. This storage area, called the hardware status area (HSA), is 24 bytes in length.

You must align the HSA on a fullword boundary. The HSA has the following format:

```
         ALIGN     WORD              ALIGN ON FULLWORD BOUNDARY
HSA      EQU       *
HSAPSW   DC        F'0'              PROGRAM STATUS WORD
HSALSB   EQU       *                 11 WORD LEVEL STATUS BLOCK
HSAIAR   DC        F'0'              INSTRUCTION ADDRESS REGISTER
HSAAKR   DC        F'0'              ADDRESS KEY REGISTER
HSALSR   DC        F'0'              LEVEL STATUS REGISTER
HSAREGS  DC       8F'0'              GENERAL REGISTERS 0-7
```

Figure 25. Format of the hardware status area (HSA)

The contents of the various HSA locations (for example PSW and AKR) contain, upon entry to your routine, the values that were in the corresponding hardware registers at the time of the exception. Also, general register 1 contains the starting instruction address (SIA) of your routine. General register 2 contains the address of your task's TCB. Your routine can examine this status information to determine whether to continue or end execution. The *Problem Determination Guide* can assist you in interpreting the information returned from an exception.

Since entry to your routine is made at the Event Driven Language level, the contents of the remaining general registers are dependent upon what instructions your program executed when the exception occurred.

Creating Your Own Task Error Exit Routine *(continued)*

Sample Task Error Exit Routine

An example of a task error exit routine follows. The sample program examines the processor status word (PSW) for the type of exception and displays the contents of some selected fields upon the loading terminal.

```
         PRINT        OFF
         COPY         PROGEQU
         PRINT        ON
         ENTRY        TSKEXIT
ERRXT    PROGRAM      MAIN=NO
TSKEXIT  EQU          *
         ALIGN        WORD
TEECB    EQU          *             TASK ERROR EXIT CONTROL BLOCK
TEECTL   DC           X'0002'       NUMBER OF DATA WORDS IN TEECB
TEESIA   DC           A(EXITRTN)    ADDRESS OF ERROR EXIT ROUTINE
TEEHSA   DC           A(HSA)        ADDRESS OF HARDWARE STATUS AREA
         ALIGN        WORD
HSA      EQU          *             HARDWARE STATUS AREA
HSAPSW   DC           F'0'          PROGRAM STATUS WORD
HSALSB   EQU          *             11 WORD LEVEL STATUS BLOCK
HSAIAR   DC           F'0'          INSTRUCTION ADDRESS REGISTER
HSAAKR   DC           F'0'          ADDRESS KEY REGISTER
HSALSR   DC           F'0'          LEVEL STATUS REGISTER
HSAREGS  DC          8F'0'          GENERAL REGISTERS 07
PCHKPLP  DATA         F'0'          PGM LOAD POINT
FAILADDR DATA         F'0'          FAILING ADDR
ADDRTBL  EQU          *
         DC           A(BIT0)
         DC           A(BIT1)
         DC           A(BIT2)
         DC           A(BIT3)
         DC           A(BIT4)
         DC           A(BIT5)
         DC           A(BIT6)
         DC           A(BIT7)
         DC           A(BIT8)
         DC           A(BIT9)
         DC           A(BIT10)
         DC           A(BIT11)
         DC           A(BIT12)
         DC           A(BIT13)
         DC           A(BIT14)
         DC           A(BIT15)
```

Figure 26 (Part 1 of 2). Sample task error exit routine

```
PSWTBL   EQU         *
BIT0     TEXT        'SPECIFICATION CHECK'
BIT1     TEXT        'INVALID STORAGE ADDRESS'
BIT2     TEXT        'PRIVILEGE VIOLATE'
BIT3     TEXT        'PROTECT CHECK'
BIT4     TEXT        'INVALID FUNCTION'
BIT5     TEXT        'FLOATING POINT EXCEPTION'
BIT6     TEXT        'STACK EXCEPTION'
BIT7     TEXT        'BIT 7 NOT USED'
BIT8     TEXT        'STORAGE PARITY CHECK'
BIT9     TEXT        'BIT 9 NOT USED'
BIT10    TEXT        'CPU CONTROL CHECK'
BIT11    TEXT        'I/O CHECK'
BIT12    TEXT        'SEQUENCE INDICATOR'
BIT13    TEXT        'AUTO IPL'
BIT14    TEXT        'TRANSLATOR ENABLED'
BIT15    TEXT        'POWER/THERMAL WARNING'
BITCNT   DATA        F'0'
PSWORK   DATA        F'0'
MSGREC   TEXT        LENGTH=80
EXITRTN  EQU         *
         TCBGET      PCHKPLP,$TCBPLP            GET PGM LOAD PT
         SUBTRACT    HSAREGS+2,PCHKPLP,RESULT=FAILADDR  FAIL ADDR
         MOVE        #1,PCHKPLP
         PRINTEXT    '@PROGRAM NAME = '
         PRINTEXT    ($PRGNAM,#1)               PRINT PGM NAME
         PRINTEXT    '@PSW = '
         PRINTNUM    HSA,MODE=HEX               PRINT HSA VALUE
         PRINTEXT    '@IAR = '
         PRINTNUM    HSA+2,MODE=HEX             PRINT INST ADDR REG
         PRINTEXT   '@PSW ANALYSIS: @'
         MOVE        PSWORK,HSAPSW
         MOVEA       #1,ADDRTBL                 MOVE MSG LIST ADDR
         DO          16,TIMES,INDEX=BITCNT
           IF        (BITCNT,GT,1)
             SHIFTL  HSAPSW,1
           ENDIF
           IF        (HSAPSW,LT,0)
             MOVE    PSWMSG,(0,#1)              POINT TO ERR MSG
             PRINTEXT MSGREC,P1=PSWMSG,SKIP=1
           ENDIF
           ADD       #1,2                       INCREMENT INDEX
         ENDDO
         PROGSTOP
         END
```

Figure 26 (Part 2 of 2). Sample task error exit routine

Creating Your Own Task Error Exit Routine *(continued)*

You must compile the task error exit routine and link-edit the assembled output with the main task. Specify the entry point name of the routine on the ERRXIT= operand of the main task.

An example of the main task that specifies the previous routine follows:

```
MAINPGM  PROGRAM   START,ERRXIT=TSKEXIT
         EXTRN     TSKEXIT
START    EQU       *
         •
         •
         •
         PROGSTOP
         ENDPROG
         END
```

Considerations on the Use of Task Error Exit Routines

You should understand the following items when you use a task error exit routine:

- A task error exit routine is a part of the task it serves. The supervisor passes control to it at the task level; it is not a subroutine of the supervisor's error handler.
- If your main program attaches multiple tasks, you should specify the ERRXIT= operand on each TASK statement.
- The registers (including the EDL software registers #1 and #2) used by the error exit routine are those normally used by the task.
- To resume task execution after the task error exit routine, you must issue a branch instruction (for Series/1 assembler) or a GOTO instruction (for EDL) to the appropriate location.
- If the task error exit routine is unable to recover from the exception, it should issue a PROGSTOP instruction.

What Happens When an Exception Occurs?

If an exception (machine check, program check, or soft exception trap) occurs during the execution of your task and you have specified a task error exit, the supervisor locates your TEECB. It then uses the TEEHSA pointer to locate your HSA and stores the hardware status information in it. Next, the supervisor retrieves the TEESIA pointer and sets it to zero to prevent recursive exceptions. Finally, the supervisor starts your task at the address it retrieved if that address is nonzero. If the TEESIA is zero or an exception occurs during any of this processing (if, for example, the TEECB is invalid), the supervisor treats the error as if you did not specify a task error exit routine. Note that even if the TEESIA is zero, the supervisor still attempts to store the hardware status.

Since the supervisor zeroes TEESIA prior to starting your task, your task error exit routine only gets control on the first exception that occurs, unless your routine restores TEESIA to its original condition. Zeroing TEESIA allows the supervisor to handle exceptions that occur in task error exit routines, thus preventing recursion in the error handling process. When you write a task error exit routine, do not restore TEESIA until the error exit routine has completed.

Notes

Chapter 5. Running Programs and Initialization Routines at IPL

You can design your system so that your programs and initialization routines are run as part of the IPL process. You can do this by:

- naming your program $INITIAL.
- creating a program named $PROG1 linked with the supervisor.
- coding the INITMOD operand on the SYSTEM statement.

Using $INITIAL to run programs at IPL is the simplest method. Programs invoked through this method do not require link-editing with the supervisor. As a result, the programs loaded can reside on disk.

When you use $PROG1 or specify initialization routines on the INITMOD operand, you must link-edit these routines to the supervisor during system generation.

The programs or routines that run could perform various functions. For example, using $INITIAL, you could have the session manager loaded in a particular partition and printer spooling in another.

Assume your Series/1 has no disk/diskette but communicates with a host over a BSC line. The host could IPL the Series/1 by transmitting the supervisor (with $PROG1). $PROG1 would run after IPL.

If you always run a program that sets up an area of storage to some value, you could specify this program as an initialization routine. You do this by coding the INITMOD operand on the SYSTEM statement.

Running Programs and Initialization Routines at IPL

This chapter describes how you can supply programs and routines to be run at IPL using either of these methods.

How to Specify $INITIAL Programs

To have your programs loaded at IPL, you must name a program $INITIAL. Two ways you can assign the name $INITIAL to a program are as follows:

- Using $DISKUT1 to rename (RE command) an existing program.
- Specifying the name $INITIAL as your program name when you prepare the program using $UPDATE or $EDXLINK.

The $INITIAL program must reside on the IPL volume.

Your $INITIAL program can issue LOADs to other programs. You have complete control of the function performed by this program.

After all system and user-written initialization routines execute, the supervisor issues a LOAD for $INITIAL.

Things You Should Know About $INITIAL

Effectively, you can use any program as a $INITIAL program. However, consider the following when you create a $INITIAL program:

- You cannot use the "??" option to specify data sets (DS=) or overlays (PGMS=) on the PROGRAM statement.
- No "program load" message is displayed when $INITIAL is loaded.
- Any errors that occur when $INITIAL is loaded are not displayed; you should check all return codes.
- If you want to prevent the supervisor from loading $INITIAL, rename the program using $DISKUT1.
- You can use the INITPRT operand of the SYSTEM statement to specify the partition into which $INITIAL is loaded.
- You can code the PARM= operand on the PROGRAM statement to receive a parameter at load time. The system passes a 1-word parameter that indicates the type of IPL — manual or auto.

How to Specify $INITIAL Programs *(continued)*

Sample $INITIAL Programs

The following examples show some of the functions you could use for $INITIAL:

Loading Programs in Three Partitions

The following sample program loads three programs. The session manager is loaded in partition 1, printer spooling in partition 2, and Indexed Access Method in partition 3. The return code is checked for load errors.

```
INIT       PROGRAM    LOADPGM
LOADPGM    EQU        *
L1         LOAD       $SMMAIN,PART=1,ERROR=NOSMGR
L2         LOAD       $SPOOL,PART=2,ERROR=NOSPL
L3         LOAD       $IAM,PART=3,ERROR=NOIAM
           GOTO       ALLDONE
NOSMGR     MOVE       RCODE,INIT
           PRINTEXT   '@LOAD ERROR FOR $SMMAIN,  RC= '
           PRINTNUM   RCODE
           GOTO       L2                  NEXT LOAD
NOSPL      MOVE       RCODE,INIT
           PRINTEXT   '@LOAD ERROR FOR $SPOOL,  RC= '
           PRINTNUM   RCODE
           GOTO       L3                  NEXT LOAD
NOIAM      MOVE       RCODE,INIT
           PRINTEXT   '@LOAD ERROR FOR $IAM,  RC= '
           PRINTNUM   RCODE
ALLDONE    PROGSTOP
RCODE      DATA       F'0'
           ENDPROG
           END
```

Determining the Type of IPL

The following sample code shows how you can determine the type of IPL based on the IPL Mode switch setting. The system passes the parameter upon IPL. Your $INITIAL program could decide what routine to invoke based on the parameter value. A zero indicates manual IPL; a one indicates auto IPL. You must code the PARM operand on the PROGRAM statement to receive this parameter. Your program must refer to this parameter as $PARM1.

If, for example, your system had an external battery-operated clock (connected via a digital input feature) or kept the date and time on a disk data set, the program could read the time and date upon an auto IPL. $INITIAL could then load the time and date into the system time and date table ($TIMRTBL).

How to Specify $INITIAL Programs *(continued)*

The following example shows how you could read the time and date from disk. The time is set to 13:24:05 and the date to December 25, 1983.

```
INIT      PROGRAM   START,PARM=1,DS=((TIMDAT,MYVOL))
          COPY      PROGEQU              RESOLVE $TIMRTBL REFERENCE
START     EQU       *
          IF        ($PARM1,EQ,1),GOTO,AUTOIPL
MANIPL    PRINTEXT  '@MANUAL IPL DONE...'
          •
          •        (routine for manual IPL)
          •
          GOTO      EXIT
AUTOIPL   EQU       *
          PRINTEXT  '@AUTO IPL DONE...'
          READ       DS1,TIMRDATA        READ TIME/DATE FROM DISK
          •
          •
          MOVE       #1,$TIMRTBL,FKEY=0
          MOVE       (8,#1),TIMRDATA,6,TKEY=0   LOAD TIME/DATE
          •
          •
          •
EXIT      PROGSTOP
TIMRDATA  DC         X'000D'              HOUR
          DC         X'0018'              MINUTE
          DC         X'0005'              SECOND
          DC         X'000C'              MONTH
          DC         X'0019'              DAY
          DC         X'0053'              YEAR
          ENDPROG
          END
```

Notes:

1. Under $EDXASM, you must include a COPY PROGEQU statement to resolve the reference to $TIMRTBL.

2. TIMRDATA is a 6-word table containing the time and date in hexadecimal.

How to Use $PROG1 at IPL

You can have an application program run at IPL by link-editing it with the supervisor. Doing this makes your program always resident in storage. Using $PROG1 could be useful if your system does not have a disk or diskette device from which to load programs.

After all system and user-written initialization routines execute, the supervisor issues an ATTACH for a $PROG1.

To use $PROG1, you must code the program as follows. The program must contain a CSECT statement with a label name of $PROG1.

```
$PROG1   CSECT
         •
         •     (source code)
         •
         PROGSTOP
         ENDPROG
         END
```

After you assemble your program, you must link-edit the assembled output with the supervisor. If you performed a tailored system generation, edit the data set that defines the supervisor modules currently in your supervisor (normally LINKCNTL on EDX002). Otherwise, you edit $LNKCNTL. An INCLUDE statement for $PROG1 on volume XS4002 exists in the link-control data set. You must blank out the asterisk preceding the INCLUDE statement and indicate on which volume your $PROG1 resides.

An example of the link-control data set with an INCLUDE statement for $PROG1 (on volume USRVOL) follows:

```
   •
   •
   •
*----------------------------------------------------------------------
*  SYSTEM INITIALIZATION - MUST BE IN PARTITION 1
*----------------------------------------------------------------------
 INCLUDE $PROG1,USRVOL    *22*   USER MODULE INCLUDED IN NUCLEUS GEN
*INCLUDE IO1024           *21*   1024 IPL SUPPORT
   •
   •
   •
```

After changing the INCLUDE statement, save the edited data set in LINKCNTL on EDX002. Next, you load $JOBUTIL and specify SUPPREPS when prompted for a data set. SUPPREPS will generate a new supervisor containing your $PROG1 program.

After you receive a -1 completion code, load $INITDSK and issue the II command to point to the new supervisor. IPL the new supervisor.

How to Use $PROG1 at IPL *(continued)*

What Happens When $PROG1 Executes?

When the supervisor attaches $PROG1, all of the storage in partition 1 is assigned to $PROG1. If you issue the $A operator command, the system will show $PROG1 in storage. Because all of partition 1 is assigned to $PROG1, you cannot load any other programs until $PROG1 issues a PROGSTOP.

How to Specify Initialization Routines

You can supply initialization routines that are run as part of the IPL. These routines are invoked after the system initialization routines execute. This section describes how you can do this.

Designing and Coding the Routine

The routine you supply can be written in EDL or Series/1 assembler. However, the first instruction of the routine must be an EDL instruction. You must also consider the following:

- The routine must be written to receive and return control in EDL.
- You must use the USER instruction to switch from EDL to assembler.
- You must preserve the contents of register 2.
- You must preserve the task control block (TCB) pointer.
- LOAD and PROGSTOP instructions are not allowed.
- Upon exit, the routine must return control to the label INITEXIT. INITEXIT is an entry point in the supervisor.

The following coding examples show how you should code your routine. The first example uses EDL only; the second uses EDL and Series/1 assembler.

How to Specify Initialization Routines *(continued)*

Routine using EDL

```
INITRTN  PROGRAM   MAIN=NO
         EXTRN     INITEXIT
         ENTRY     INIT
INIT     EQU       *
         •
         •     (EDL code)
         •
         GOTO      INITEXIT
```

Routine using EDL and Series/1 Assembler

```
INITRTN  CSECT
         EXTRN     INITEXIT
         USER      INIT
INIT     EQU       *
         •
         •     (assembler code)
         •
         MVA       INITEXIT,R1
         BX        CMDSETUP              BACK TO EDL
```

Link-Editing the Routine with the Supervisor

After you assemble your routine, you must link-edit the assembled output with the supervisor. If you performed a tailored system generation, edit the data set that defines the supervisor modules currently in your supervisor (normally LINKCNTL on EDX002). Otherwise, you edit $LNKCNTL. Insert an INCLUDE statement specifying the name of the assembled output in the area designated for user initialization modules. For example, if your assembled output module is named INITOBJ on volume MYVOL, the INCLUDE statement would be as follows:

```
   •
   •
   •
*-----------------------------------------------------------------
*  INSERT USER INITIALIZATION MODULES HERE
*-----------------------------------------------------------------
 INCLUDE INITOBJ,MYVOL        YOUR NEW INIT ROUTINE
   •
   •
   •
```

After inserting the new INCLUDE statement, save the edited data set in LINKCNTL on EDX002. Optionally, you can include the initialization routine as an overlay to save storage.

How to Specify Initialization Routines *(continued)*

The *Installation and System Generation Guide* describes how to specify and use the overlay feature. If you do not use the overlay feature, go to the section "Specifying the Routine on the SYSTEM Statement" on page CU-62.

Specifying the Routine on the SYSTEM Statement

You must edit the data set which defines your system to specify the routine. This data set is normally $EDXDEFS on volume EDX002. Code the INITMOD operand on the SYSTEM statement to specify the entry point name of your routine. You can specify one or more routines. If you do, specify each entry-point name separated by a comma and enclose the name list in parentheses. The routines are executed in the order you specify.

An example of the SYSTEM statement with the INITMOD operand coded follows. Two initialization routines are specified.

```
         SYSTEM    STORAGE=64,INITMOD=(INIT,RTNA)
           •
           •
           •
```

After you edit and save $EDXDEFS, load $JOBUTIL and specify SUPPREPS when prompted for a data set. SUPPREPS will generate a new supervisor containing your initialization routine.

Upon receiving a -1 completion code, load $INITDSK and issue the II command to point to the new supervisor. IPL the new supervisor.

Chapter 6. Adding Your Own Device Support

If you have a need to use a device or device feature not supported under EDX, you can provide support for that device or feature through the use of EXIO. The system's EXIO support enables you to control, from your programs, any device that meets the hardware channel architecture (such as plug compatibility and device control blocks) of the Series/1. These devices can be IBM or original equipment manufacturer (OEM) devices.

This chapter describes how you can provide your own device support using EXIO. In addition, a sample program using EXIO is shown. The sample program illustrates an approach you could use to support a device attached to the 2095/2096 Feature Programmable Multiline Controller/Adapter using expanded mode (with continuous receive) and one stop bit.

[illegible]

The system's EXIO support enables you to do I/O level programming for a device attached to the Series/1. Further, with EXIO you can do the following:

- Gain closer control of an EDX-supported device. With EXIO, you control every aspect of the device's operation. For example, you can provide a more extensive error-handling and error-recovery procedure than EDX provides for that device.
- Issue I/O from a program in any partition.
- Provide support for a device without adding any new supervisor code. The device support resides in your program.

How You Can Use EXIO *(continued)*

- Write the support as reentrant code or as subroutines you link to each program using the device(s).

- Provide I/O level programming in EDL without using Series/1 assembler. However, some device operations may require the speed of execution that Series/1 assembler provides. You can mix the two languages and assemble with $S1ASM.

The next section discusses several considerations you need to think about before you implement the device support. The topics presented can assist you when you actually start writing the device support code.

Planning for Your Device Support

Because you must control every operation the device performs when you use EXIO, you *must* be familiar with the device you intend to support. The *IBM Series/1 Principles of Operation*, GA34-0152 manual presents a general overview of the Series/1 I/O architecture.

The following topics describe some of the device requirements with which you should be familiar.

Do You Understand the Hardware Control Block Functions?

To properly control the device, you must understand the function of the hardware control blocks. In particular, you must understand the immediate device control block (IDCB) and the optional device control block (DCB). These control blocks contain the I/O operation code and other information the attachment needs to issue I/O to the device.

The hardware description manual for the device or attachment you support normally contains information on these control blocks and how you use them.

What Types of Device Interrupts Should You Plan For?

If the device produces interrupts, your device support must supply all required information needed to service the interrupts. In addition, your device support must prepare the device for interrupts as well as disable interrupts when the task ends.

You would typically have separate tasks in your program to handle device interrupts and post events.

Normally, you obtain information on device interrupts from the hardware device description manual.

Planning for Your Device Support (continued)

Does the Device Have Any Special Timing Considerations?

You must determine if your device has any unique timing requirements. For example, the amount of time in which an interrupt must be serviced or a data transfer completed. If timing is critical for the device, you may have to establish task priorities. You may also have to consider performance differences using EXIO versus Series/1 assembler code.

Do You Have to Detect and Handle Errors?

The attachment reports status at the start of and after the completion of an I/O operation. This information is returned as status words and condition codes. You must design your device support to detect and handle any errors it encounters.

All possible error conditions should be described in the hardware device description manual.

The device description manual describes the possible errors you could encounter and how they are reported.

How Many Devices Will You Support?

The number of devices you support may determine how you design the support. Normally, if you only support one device from one program, the EXIO code and much of the data and device control information can reside in that program.

When you support multiple devices, you must provide a copy of the data and device control information for each device.

How Many Applications Will Use the Device?

If multiple applications will request the use of the support at the same time, you must serialize the support's use. You provide serial use through the ENQ/DEQ instructions. Further, if these applications reside in different partitions, you must use the system's cross-partition services to move data and device control information across the partitions.

Do You Have to Initialize the Device?

Some attachments and/or devices require special initialization or a random access memory load prior to their use. EDX does not initialize devices you define as an EXIO device. Device initialization is your responsibility.

You must also know the engineering change (EC) level of your device. Different device EC levels may require that you select from various random access memory load modules at initialization. The EC level and initialization code must match for the device.

Defining the Device at System Generation

You use the EXIODEV statement to define your device at system generation. The device you define must not be defined in the system by any other configuration statement.

If your device support performs cycle steal operations or requires chained DCBs to complete an operation, you must specify the MAXDCB= operand. In addition, cycle steal operations return residual status information. You must specify the RSB= operand to indicate the number of residual status bytes returned from the operation.

The EXIODEV statement is discussed in the *Installation and System Generation Guide.*

The supervisor must also contain EXIO support modules. You must specify INCLUDE statements for the modules IOSEXIO and EXIOINIT in your link control data set.

This section explains a sample program that uses EXIO to control a device. The 3101 Model 1 terminal (character mode) is the device used and is connected to the 2095/2096 Feature Programmable multiline attachment. The program provides support for expanded mode (with continuous receive) and one stop bit during data transmission.

Controlling a device with continuous receive enables a receive channel for the device to be open at all times. You would use this feature under EXIO when a device requires input at a speed at which EDL terminal I/O instructions cannot provide.

The sample program, when loaded, prompts for input, loops to receive ten lines of input, and prints the input on the printer.

The instructions and statements the program uses to perform I/O operations to the device are: EXIO, EXOPEN, IDCB, and DCB. Refer to the *Language Reference* for the coding syntax and description of these instructions and statements.

The EXIODEV statement for this device follows:

```
         EXIODEV    ADDRESS=60,MAXDCB=1,RSB=6,END=YES
```

As with any support you provide using EXIO, you must understand the characteristics of the device or attachment. The *IBM Series/1 Communications Features Description*, GA34-0028 can assist you in understanding the I/O operations to the attachment used in the sample program.

Writing the EXIO Code (continued)

Preparing the Device for Interrupts

Before the program issues any I/O operations to the device, it must initiate all interrupt handling tasks, open the device, and prepare the device for interrupts.

The interrupt handling tasks are separate tasks which the (main) program attaches. Each task waits for the hardware to post an ECB indicating an interrupt has occurred. When the hardware posts the ECB, the task does some processing and posts an ECB in the main program to indicate the interrupt has been serviced. After the task posts the main program, the interrupt handling task waits again for the next interrupt.

The interrupt handling tasks in this program service the following interrupts:

- Device end interrupts
- Controller end interrupts
- Exception interrupts.

The descriptions and code for the interrupt handling tasks follow:

This program uses the task DEVINT to wait on and service device end interrupts. A device end interrupt indicates that the device was able to successfully complete the program's I/O request.

This task waits for the hardware to post the event control block DEVEND. The main program waits for this task to post DONEECB.

The code that handles device end interrupts follows:

```
DEVINT   TASK      DEVSTART
DEVSTART WAIT      DEVEND                 WAIT FOR DEVICE END INTERRUPT
         RESET     DEVEND
         POST      DONEECB,-1
         GOTO      DEVSTART
         ENDTASK
```

Writing the EXIO Code *(continued)*

Controller End Interrupt Task

This program uses the task ENDINT to wait on and service controller end interrupts. A controller end interrupt indicates that the attachment can now accept an I/O request (no longer busy).

This task waits for the hardware to post the event control block CENDECB. The main program waits for this task to post CTLREND.

The code that handles controller end interrupts follows:

```
ENDINT    TASK      CTLSTART
CTLSTART  WAIT      CENDECB          WAIT FOR CONTROLLER END INTERRUPT
          RESET     CENDECB
          POST      CTLREND,-1
          RESET     CTLREND
          GOTO      CTLSTART
          ENDTASK
```

Exception Interrupt Task

This program uses the task EXCINT to wait on and service exception interrupts. An exception interrupt indicates that the device was unable to perform the I/O request successfully.

When an exception occurs, this task examines the hardware status information and prints the information on the printer.

This task also examines word 1, bit 15 of the cycle steal status. When bit 15 is on, a buffer overrun condition exists. This task signals a buffer overrun condition by posting DONEECB with a value of 2. The main program must then issue a "read adapter buffer" operation.

The code that handles exception interrupts follows:

```
EXCINT   TASK      EXCSTART
EXCSTART WAIT      EXCEPT                WAIT FOR EXCEPTION INTERRUPT
         RESET     EXCEPT
         IF          (INTWORD,EQ,X'A0',BYTE),THEN    SHORT RECORD
           POST      DONEECB,-1                      POST GOOD RETURN
         ELSE
           IF        (INTWORD,EQ,X'20',BYTE),THEN    LONG RECORD
             PRINTEXT  '@LONG RECORD@'
           ELSE
             IF      (INTWORD,EQ,X'80',BYTE),AND,((SCSSDATA+2),EQ,     C
               X'40',BYTE),THEN                      TIME-OUT
               PRINTEXT  '@TIME-OUT@'
             ELSE
               PRINTEXT  '@OTHER EXCEPTION INTERRUPT, '
             ENDIF
           ENDIF
          ENQT       $SYSPRTR
           PRINTEXT  'CSS = '
           PRINTNUM  SCSSDATA,3,MODE=HEX          CYCLE STEAL STATUS
           PRINTEXT  '@INTWORD,LSR,ECB ADDR : '
           PRINTNUM  INTWORD,3,MODE=HEX
           PRINTEXT  SKIP=1
           DEQT
           MOVE      WD1,SCSSDATA+2
           SHIFTL    WD1,15                          ISOLATE BIT 15
           IF        (WD1,EQ,X'8000')                BIT 15 = 1 ?
                POST   DONEECB,2          INDICATE READ ADAPTER BUFFER
                GOTO   EXCSTART
           ENDIF
           POST      DONEECB,1                      POST ERROR RETURN
         ENDIF
         GOTO      EXCSTART
         ENDTASK
```

Writing the EXIO Code *(continued)*

After the program attaches the interrupt handling tasks, the program opens and prepares the device. The code that performs these functions follows:

```
EXIOREC  PROGRAM    EXSTART
EXSTART  EQU        *
         ATTACH     DEVINT          DEVICE END INTERRUPT HANDLING TASK
         ATTACH     EXCINT           EXCEPTION INTERRUPT HANDLING TASK
         ATTACH     ENDINT      CONTROLLER END INTERRUPT HANDLING TASK
         EXOPEN     60,INTWORK,ERROR=OPENERR            OPEN BASE LINE
         EXIO       PREIDCB,ERROR=PREPERR             ENABLE INTERRUPT
         PRINTEXT   '@DEVICE OPEN AND PREPARED@'
           •
           •
           •
         CALL       SETMODE
```

Next the program must establish the mode of transmission. The next section explains how this is done.

[illegible]

The program calls a subroutine (SETMODE) to establish the transmission mode. SETMODE establishes the transmission mode as being expanded mode (with continuous receive) using one stop bit.

The code for the SETMODE subroutine follows:

```
         SUBROUT    SETMODE
         EXIO       RESET                        DEVICE RESET
*
*                    ISSUE SET MODE DCB TO CHANGE
*                    NUMBER OF STOP BITS TO ONE
*
         RESET      DONEECB
         EXIO       SETIDCB,ERROR=SETERR
         WAIT       DONEECB
*
*                    ISSUE SET EXPANDED MODE DCB
*                    TO SET CONTINUOUS RECEIVE
*
         RESET      DONEECB
         EXIO       EXPIDCB,ERROR=EXPERR
         WAIT       DONEECB
         RETURN
```

SETIDCB is the label of an IDCB statement and points to the label of the DCB statement, SETDCB. These two statements define one stop bit:

```
SETIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=SETDCB
*                     DEVMOD SETUP FOR SET MODE 1 STOP BIT
*                         9600BPS=07    CR=0D    LF=0A
SETDCB    DCB       DEVMOD=B4,DVPARM1=070D,DVPARM2=0A00
```

On the DCB statement, the value for the DEVMOD= operand is B4. This value sets word 0 (bits 8–15) of the device control block to the binary value 10110100. These bit settings indicate the following:

- Set mode
- Asynchronous operation
- Eight bits per character
- One stop bit
- Odd parity
- Parity disabled.

EXPIDCB is also the label of an IDCB statement and points to the label of the DCB statement, EXPDCB. These two statements define expanded mode with continuous receive:

```
EXPIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=EXPDCB,MOD4=C
*                     SET CONTINUOUS RECV MODE ** 15 BYTE BUFFER   **
*                                              * IN DEVICE ADAPTER *
EXPDCB    DCB       DEVMOD=01,DVPARM3=0001
```

Note that the operand MOD4=C is coded on the IDCB statement. This operand alters the IDCB and requests a "start control" operation.

The DVPARM3=0001 operand on the DCB statement sets word 3 (bit 15) of of the device control block to indicate continuous receive.

After the program establishes the mode of transmission, the program writes a prompt message to the terminal. This sequence is described next.

Writing the EXIO Code *(continued)*

Writing Data to the Terminal

The program requests input by writing the message "ENTER DATA:" to the terminal. After writing the message, the program checks for a -1 return code and also a controller (attachment) busy condition.

Note: For this program, only one port on the attachment is active, however, if multiple ports were active, a controller busy condition could occur. This program detects and handles controller busy conditions.

If the controller is busy when the program issues an I/O request to the device, the EXIO operation fails. When the EXIO operation fails, you must reset the attachment. However, the reset also resets the continuous receive. The program calls the SETMODE subroutine to reenable continuous receive.

The code for the program at this point looks like the following:

```
EXIOREC  PROGRAM   EXSTART
EXSTART  EQU       *
         ATTACH    DEVINT          DEVICE END INTERRUPT HANDLING TASK
         ATTACH    EXCINT           EXCEPTION INTERRUPT HANDLING TASK
         ATTACH    ENDINT     CONTROLLER END INTERRUPT HANDLING TASK
         EXOPEN    60,INTWORK,ERROR=OPENERR            OPEN BASE LINE
         EXIO      PREIDCB,ERROR=PREPERR             ENABLE INTERRUPT
         PRINTEXT  '@DEVICE OPEN AND PREPARED@'
         CALL      SETMODE
LOOP1    EQU       *
*
*                   ISSUE TRANSMIT END DCB
*                   TO WRITE MESSAGE TO TERMINAL
*
         RESET     DONEECB
WRITE    EXIO      WR1IDCB                TRANSMIT END
         MOVE      RC,EXIOREC
         IF        (RC,EQ,7)              TEST FOR CONTROLLER BUSY
           WAIT      CTLREND
           CALL      SETMODE
           GOTO      WRITE
         ENDIF
         IF        (RC,NE,-1),GOTO,WRERR
         WAIT      DONEECB                WAIT FOR COMPLETION OF WRITE
         IF        (DONEECB,NE,-1),THEN           CHECK FOR GOOD WRITE
*          INSERT USER ERROR ROUTINE
         ENDIF
         •
         •
         •
```

The IDCB statement for WR1IDCB points to the DCB labeled WR1DCB. This DCB contains the address of the message data (WRDATA). The message data is ASCII code and is 16 bytes in length.

The IDCB and DCB statements for the write operation follow:

```
WR1IDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=WR1DCB
WR1DCB    DCB       DEVMOD=01,DVPARM2=0003,COUNT=16,DATADDR=WRDATA
*                      TIMER1=10MS
```

The following code defines the message data area:

```
*
WRDATA    DATA      X'0D0A'          CR/LF
          DATA      X'454E'          EN
          DATA      X'5445'          TE
          DATA      X'5220'          R
          DATA      X'4441'          DA
          DATA      X'5441'          TA
          DATA      X'203A'            :
          DATA      X'2020'
```

The next section describes how the program reads input data from the terminal.

Reading Data from the Terminal

The program sets up to do a read operation (with time-out) by issuing an EXIO instruction to the IDCB labeled RD1IDCB. The DCB associated with this read operation indicates 12 bytes of data will be stored beginning at address REDATA.

The IDCB and DCB statements for the read operation follow:

```
RD1IDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=RD1DCB
RD1DCB    DCB       IOTYPE=INPUT,DEVMOD=05,DVPARM2=1000,COUNT=12,        C
                DATADDR=REDATA
*                     TIMER1=13.6SEC
```

The program enters a DO loop that reads a line of input and writes the input (REDATA) to the printer. The program loops 10 times and then prompts for input again. If during the loop you enter "END," the program ends.

Also within the loop, the program checks for a "buffer overrun" condition. The program indicates a buffer overrun condition when DONEECB equals 2. The program calls the RDBUFF subroutine to handle buffer overrun conditions.

The code to perform the read operation within the DO loop follows:

```
         •
         •
         •
         DO         10,TIMES
         MOVE       REDATA,C' ',(40,BYTES)
*                    ISSUE RECEIVE WITH TIME-OUT DCB
*                    TO READ DATA FROM TERMINAL
*
         RESET      DONEECB
READ     EXIO       RD1IDCB                 RECEIVE WITH TIME-OUT
         MOVE       RC,EXIOREC
         IF         (RC,EQ,7)               TEST FOR CONTROLLER BUSY
           WAIT       CTLREND
           CALL       SETMODE
           GOTO       READ
         ENDIF
         IF         (RC,NE,-1),GOTO,RDERR
         WAIT       DONEECB                 WAIT FOR COMPLETION OF READ
         IF         (DONEECB,EQ,2)
              CALL    RDBUFF
              GOTO    RDEND
         ENDIF
         IF         (DONEECB,NE,-1),THEN               CHECK FOR GOOD READ
*          INSERT USER ERROR ROUTINE
         ENDIF
         ENQT       $SYSPRTR
         PRINTEXT   '@INPUT DATA FROM TERMINAL: '
         PRINTNUM   REDATA,10,MODE=HEX
         PRINTEXT   SKIP=1
         DEQT
RDEND    EQU        *
         IF         (REDATA,EQ,ENDDATA,3),GOTO,END    TEST FOR ''END''
         ENDDO
         GOTO       LOOP1
END      PROGSTOP
         •
         •
         •
```

[illegible]

[illegible]

The RDBUFF subroutine performs a "read adapter buffer" operation followed by a "start cycle steal status" operation. Both operations must be done to reset a buffer overrun condition.

The RDBUFF subroutine follows:

```
SUBROUT   RDBUFF              SUBRTN FOR BUFFER OVERRUN
RESET     DONEECB
EXIO      RDAIDCB,ERROR=RABERR
WAIT      DONEECB              WAIT FOR COMPLETION OF WRITE
PRINTEXT  'CC = '              PRINT COMPLETION CODE
PRINTNUM  DONEECB
PRINTEXT  SKIP=1
ENQT      $SYSPRTR
PRINTEXT  '@READ ADAPTER BUFFER: '
PRINTNUM  REDATA,10,MODE=HEX
PRINTEXT  SKIP=1
DEQT
RESET     DONEECB
EXIO      CSSIDCB,ERROR=CSSERR
PRINTEXT  '@READ CYCLE STEAL STATUS DCB ISSUED, '
WAIT      DONEECB
PRINTEXT  'CC = '              PRINT COMPLETION CODE
PRINTNUM  DONEECB
PRINTEXT  SKIP=1
RETURN
```

Writing the EXIO Code *(continued)*

Reporting Error Return Codes

All EXIO programs should do extensive error checking and reporting. Use the ERROR= operand on the EXIO instruction to set up an error exit. The system passes control to the label you specify on this operand. The error exits in the sample program follow:

```
*
*                ERROR EXIT SECTION
*
OPENERR  EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@OPEN FAILED, '
         GOTO      ERREND
PREPERR  EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@PREPARE FAILED, '
         GOTO      ERREND
SETERR   EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@SET MODE FAILED, '
         GOTO      ERREND
EXPERR   EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@SET EXPANDED MODE FAILED, '
         GOTO      ERREND
RABERR   EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@READ ADAPTER BUFFER FAILED, '
         GOTO      ERREND
CSSERR   EQU       *
         MOVE      RC,EXIOREC
         PRINTEXT  '@READ CYCLE STEAL STATUS FAILED, '
         GOTO      ERREND
WRERR    EQU       *
         PRINTEXT  '@WRITE ERROR, '
         GOTO      ERREND
RDERR    EQU       *
         PRINTEXT  '@READ ERROR, '
ERREND   EQU       *
         PRINTEXT  'RETURN CODE = '
         PRINTNUM  RC
         PRINTEXT  SKIP=1
         GOTO      END
```

Sample EXIO Program

The coding segments throughout this chapter showed you can create your own device support. The following is the sample program in its entirety:

```
EXIOREC  PROGRAM    EXSTART
EXSTART  EQU        *
         ATTACH     DEVINT          DEVICE END INTERRUPT HANDLING TASK
         ATTACH     EXCINT           EXCEPTION INTERRUPT HANDLING TASK
         ATTACH     ENDINT      CONTROLLER END INTERRUPT HANDLING TASK
         EXOPEN     60,INTWORK,ERROR=OPENERR            OPEN BASE LINE
         EXIO       PREIDCB,ERROR=PREPERR             ENABLE INTERRUPT
         PRINTEXT   '@DEVICE OPEN AND PREPARED@'
         CALL       SETMODE
LOOP1    EQU        *
*
*                    ISSUE TRANSMIT END DCB
*                    TO WRITE MESSAGE TO TERMINAL
*
         RESET      DONEECB
WRITE    EXIO       WR1IDCB                TRANSMIT END
         MOVE       RC,EXIOREC
         IF         (RC,EQ,7)              TEST FOR CONTROLLER BUSY
           WAIT       CTLREND
           CALL       SETMODE
           GOTO       WRITE
         ENDIF
         IF         (RC,NE,-1),GOTO,WRERR
         WAIT       DONEECB                WAIT FOR COMPLETION OF WRITE
         IF         (DONEECB,NE,-1),THEN           CHECK FOR GOOD WRITE
*          INSERT USER ERROR ROUTINE
         ENDIF
         DO         10,TIMES
         MOVE       REDATA,C' ',(40,BYTES)
*                    ISSUE RECEIVE WITH TIME-OUT DCB
*                    TO READ DATA FROM TERMINAL
*
         RESET      DONEECB
READ     EXIO       RD1IDCB                        RECEIVE WITH TIME-OUT
         MOVE       RC,EXIOREC
         IF         (RC,EQ,7)             TEST FOR CONTROLLER BUSY
           WAIT       CTLREND
           CALL       SETMODE
           GOTO       READ
         ENDIF
         IF         (RC,NE,-1),GOTO,RDERR
         WAIT       DONEECB                   WAIT FOR COMPLETION OF READ
         IF         (DONEECB,EQ,2)
              CALL    RDBUFF
              GOTO    RDEND
         ENDIF
```

Figure 27 (Part 1 of 6). Sample EXIO program

Sample EXIO Program *(continued)*

```
         IF        (DONEECB,NE,-1),THEN              CHECK FOR GOOD READ
*          INSERT USER ERROR ROUTINE
         ENDIF
         ENQT      $SYSPRTR
         PRINTEXT  '@INPUT DATA FROM TERMINAL: '
         PRINTNUM  REDATA,10,MODE=HEX
         PRINTEXT  SKIP=1
         DEQT
RDEND    EQU       *
         IF        (REDATA,EQ,ENDDATA,3),GOTO,END     TEST FOR ''END''
         ENDDO
         GOTO      LOOP1
END      PROGSTOP
*
*  INTERRUPT TASKS
*
DEVINT   TASK      DEVSTART
DEVSTART WAIT      DEVEND                   WAIT FOR DEVICE END INTERRUPT
         RESET     DEVEND
         POST      DONEECB,-1
         GOTO      DEVSTART
         ENDTASK
*
ENDINT   TASK      CTLSTART
CTLSTART WAIT      CENDECB              WAIT FOR CONTROLLER END INTERRUPT
         RESET     CENDECB
         POST      CTLREND,-1
         RESET     CTLREND
         GOTO      CTLSTART
         ENDTASK
*
EXCINT   TASK      EXCSTART
EXCSTART WAIT      EXCEPT                    WAIT FOR EXCEPTION INTERRUPT
         RESET     EXCEPT
         IF          (INTWORD,EQ,X'A0',BYTE),THEN    SHORT RECORD
           POST      DONEECB,-1                      POST GOOD RETURN
         ELSE
           IF        (INTWORD,EQ,X'20',BYTE),THEN    LONG RECORD
             PRINTEXT  '@LONG RECORD@'
           ELSE
             IF      (INTWORD,EQ,X'80',BYTE),AND,((SCSSDATA+2),EQ,      C
               X'40',BYTE),THEN                      TIME-OUT
               PRINTEXT  '@TIME-OUT@'
             ELSE
               PRINTEXT  '@OTHER EXCEPTION INTERRUPT, '
             ENDIF
           ENDIF
```

Figure 27 (Part 2 of 6). Sample EXIO program

Sample EXIO Program *(continued)*

```
           ENQT       $SYSPRTR
            PRINTEXT  'CSS = '
            PRINTNUM  SCSSDATA,3,MODE=HEX          CYCLE STEAL STATUS
            PRINTEXT  '@INTWORD,LSR,ECB ADDR : '
            PRINTNUM  INTWORD,3,MODE=HEX
            PRINTEXT  SKIP=1
            DEQT
            MOVE      WD1,SCSSDATA+2
            SHIFTL    WD1,15                          ISOLATE BIT 15
            IF        (WD1,EQ,X'8000')                BIT 15 = 1 ?
                 POST   DONEECB,2         INDICATE READ ADAPTER BUFFER
                 GOTO   EXCSTART
            ENDIF
            POST      DONEECB,1                      POST ERROR RETURN
          ENDIF
          GOTO      EXCSTART
          ENDTASK
*
*  ERROR EXIT SECTION
*
OPENERR   EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@OPEN FAILED, '
          GOTO      ERREND
PREPERR   EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@PREPARE FAILED, '
          GOTO      ERREND
SETERR    EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@SET MODE FAILED, '
          GOTO      ERREND
EXPERR    EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@SET EXPANDED MODE FAILED, '
          GOTO      ERREND
RABERR    EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@READ ADAPTER BUFFER FAILED, '
          GOTO      ERREND
CSSERR    EQU       *
          MOVE      RC,EXIOREC
          PRINTEXT  '@READ CYCLE STEAL STATUS FAILED, '
          GOTO      ERREND
WRERR     EQU       *
          PRINTEXT  '@WRITE ERROR, '
          GOTO      ERREND
RDERR     EQU       *
          PRINTEXT  '@READ ERROR, '
ERREND    EQU       *
          PRINTEXT  'RETURN CODE = '
          PRINTNUM  RC
          PRINTEXT  SKIP=1
          GOTO      END
```

Figure 27 (Part 3 of 6). Sample EXIO program

Sample EXIO Program *(continued)*

```
*
*  SUBROUTINES
*
         SUBROUT    SETMODE
         EXIO       RESET                             DEVICE RESET
*
*                    ISSUE SET MODE DCB TO CHANGE
*                    NUMBER OF STOP BITS TO ONE
*
         RESET      DONEECB
         EXIO       SETIDCB,ERROR=SETERR
         WAIT       DONEECB
*
*                    ISSUE SET EXPANDED MODE DCB
*                    TO SET CONTINUOUS RECEIVE
*
         RESET      DONEECB
         EXIO       EXPIDCB,ERROR=EXPERR
         WAIT       DONEECB
         RETURN
*
         SUBROUT    RDBUFF              SUBRTN FOR BUFFER OVERRUN
         RESET      DONEECB
         EXIO       RDAIDCB,ERROR=RABERR
         WAIT       DONEECB              WAIT FOR COMPLETION OF WRITE
         PRINTEXT   'CC = '              PRINT COMPLETION CODE
         PRINTNUM   DONEECB
         PRINTEXT   SKIP=1
         ENQT       $SYSPRTR
         PRINTEXT   '@READ ADAPTER BUFFER: '
         PRINTNUM   REDATA,10,MODE=HEX
         PRINTEXT   SKIP=1
         DEQT
         RESET      DONEECB
         EXIO       CSSIDCB,ERROR=CSSERR
         PRINTEXT   '@READ CYCLE STEAL STATUS DCB ISSUED, '
         WAIT       DONEECB
         PRINTEXT   'CC = '              PRINT COMPLETION CODE
         PRINTNUM   DONEECB
         PRINTEXT   SKIP=1
         RETURN
```

Figure 27 (Part 4 of 6). Sample EXIO program

```
*
*  DATA BUFFERS
*
WRDATA   DATA      X'0D0A'          CR/LF
         DATA      X'454E'          EN
         DATA      X'5445'          TE
         DATA      X'5220'          R
         DATA      X'4441'          DA
         DATA      X'5441'          TA
         DATA      X'203A'           :
         DATA      X'2020'
REDATA   DATA      20F'0'
ENDDATA  DATA      X'454E4400'      ASCII END
SCSSDATA DATA      3F'0'            6 BYTE OF CYCLE STEAL STATUS
RC       DATA      F'0'
WD1      DATA      F'0'
*
*  INTERRUPT DEFINE INFORMATION
*
INTWORK  DC        A(INTWORD)       INTERRUPT BYTE AND ADDRESS SAVE AREA
         DC        A(INTECB)        INTERRUPT CONDITION CODE ECB
         DC        A(SCSSDCB)       START CYCLE STEAL STATUS DCB
INTWORD  DATA      F'0'             INTERRUPT STATUS / DEVICE ADDRESS
         DATA      F'0'             LSR AT TIME OF INTERRUPT
         DATA      F'0'             ADDRESS OF ECB POSTED
INTECB   DATA      A(CENDECB)       CC=0
         DATA      A(NA)            CC=1
         DATA      A(EXCEPT)        CC=2 EXCEPTION
         DATA      A(DEVEND)        CC=3 DEVICE END
         DATA      A(NA)            CC=4
         DATA      A(NA)            CC=5
         DATA      A(NA)            CC=6
         DATA      A(NA)            CC=7
```

Figure 27 (Part 5 of 6). Sample EXIO program

```
*
*  IMMEDIATE DEVICE CONTROL BLOCKS
*
RESET     IDCB      COMMAND=RESET,ADDRESS=60
PREIDCB   IDCB      COMMAND=PREPARE,ADDRESS=60,LEVEL=1,IBIT=ON
SETIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=SETDCB
EXPIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=EXPDCB,MOD4=C
WR1IDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=WR1DCB
RD1IDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=RD1DCB
RDAIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=RDADCB
CSSIDCB   IDCB      COMMAND=START,ADDRESS=60,DCB=SCSSDCB,MOD4=F
*
*  DEVICE CONTROL BLOCKS
*
SETDCB    DCB       DEVMOD=B4,DVPARM1=070D,DVPARM2=0A00
*                     DEVMOD SETUP FOR SET MODE 1 STOP BIT
*                     9600BPS=07   CR=0D   LF=0A
EXPDCB    DCB       DEVMOD=01,DVPARM3=0001
*                     SET CONTINUOUS RECEIVE MODE ** 15 BYTE BUFFER   **
*                                                  * IN DEVICE ADAPTER *
WR1DCB    DCB       DEVMOD=01,DVPARM2=0003,COUNT=16,DATADDR=WRDATA
*                     TIMER1=10MS
RD1DCB    DCB       IOTYPE=INPUT,DEVMOD=05,DVPARM2=1000,COUNT=12,       C
               DATADDR=REDATA
*                     TIMER1=13.6SEC
*
*         READ ADAPTER BUFFER DCB
*
RDADCB    DCB       IOTYPE=INPUT,DEVMOD=74,COUNT=14,DATADDR=REDATA
*
*
SCSSDCB   DCB       IOTYPE=INPUT,COUNT=6,DATADDR=SCSSDATA
*
*  EVENT CONTROL BLOCKS
*
CENDECB   ECB       0               INTERRUPT CONDITION CODE 0
EXCEPT    ECB       0               INTERRUPT CONDITION CODE 2
DEVEND    ECB       0               INTERRUPT CONDITION CODE 3
NA        ECB       0               NOT USED, PAPER WORK ONLY
DONEECB   ECB       0               OPERATION
CTLREND   ECB       0               CONTROLLER END ECB
*                             THIS ECB WILL BE WAITED ON BY ANY LINE
*                             ATTACHED TO THE CONTROLLER AT ADDRESS 60
*                             WHEN THE LINE GETS A CONTROLLER BUSY
*                             CONDITION. THE CONTROLLER END INTERRUPT
*                             WILL COME BACK ON THE BASE ADDRESS 60 FOR
*                             ANY LINE ATTACHED TO THE CONTROLLER.
***********************************************************************
         ENDPROG
         END
```

Figure 27 (Part 6 of 6). Sample EXIO program

Chapter 7. Creating Your Own EDL Instruction

If the Event Driven Language (EDL) does not provide an instruction that performs a function you need, you can create your own instruction to provide that function. This chapter explains how you can build an instruction that you can compile using $EDXASM.

The *Internal Design* provides a detailed discussion of how $EDXASM processes EDL instructions.

One of the steps to implement a new EDL instruction will require you to write some Series/1 assembler code. You will need the Series/1 Macro Assembler ($S1ASM) in that step.

Creating Your Own EDL Instruction

Defining the Instruction Requirements

The first step in creating a new instruction is defining what function the instruction is to perform. The function the instruction performs determines the coding syntax such as the use of:

- positional operands
- keyword operands
- indexable operands.

This chapter explains how to create a sample EDL instruction called NEWCMD. NEWCMD has the following characteristics:

- one positional operand
- two optional keyword operands (one of which is P1=)
- two indexable operands
- adds the value 1 to operand one, or
- adds the value of the keyword parameter to operand one
- generates a new operation code.

The system reserves two operation codes for your use — 01 and 02. The NEWCMD instruction will use 01 as the new operation code.

Using the above syntax definition, you could code NEWCMD any of the following ways:

```
LABEL1   NEWCMD   X                ADD 1 TO X
LABEL2   NEWCMD   X,KWD=Y          ADD VALUE OF Y TO X
LABEL3   NEWCMD   X,KWD=Y,P1=Z     ADD VALUE OF Y TO X
LABEL4   NEWCMD   X,KWD=(4,#1)     ADD VALUE AT (4,#1) TO X
```

After you define the function and syntax of the instruction, you must define a model of the instruction in an overlay program. This is discussed next.

Creating an Overlay Program to Build the Instruction

You define a model of the instruction in an overlay program. In addition, the overlay program contains statements and subroutines that check syntax and build object code for the new instruction.

Note: The overlay program you supply is unique to $EDXASM. Do not confuse the overlay program discussed in this chapter with EDL or $EDXLINK overlays.

A brief description of the statements you can use follows. These statements are described in detail in the section "Overlay Program Statements" on page CU-111.

$IDEF	Defines a model or prototype instruction.
ASMERROR	Generates syntax error messages.
OTE	Defines an object text element.
SLE	Defines a sublist element.

The subroutines you can use follow. These are described in detail in the section "Overlay Program Subroutines" on page CU-117.

$INDEX	Examines operands for index register usage.
BLDTXT	Builds object text from object text elements.
GETVAL	Evaluates character strings from a sublist element.
LABELS	Defines or resolves labels for symbol table entries.
MOVEBYTE	Moves a byte string to a target location.
OPCHECK	Checks instruction syntax and builds object code for each operand.
SLPARSE	Divides (parses) an input string into sublist elements.

You may use any or all of these statements and subroutines in the overlay program you create. The overlay program for the NEWCMD instruction uses $IDEF, $INDEX, ASMERROR, and OTE.

Creating Your Own EDL Instruction

Creating an Overlay Program to Build the Instruction *(continued)*

Building the Model Instruction

You use the $IDEF statement to build a model of the instruction. When you code $IDEF, you specify the positional operands and keywords of the instruction. The number of positional and keyword operands for an instruction must not exceed 50.

You can optionally specify error exits on $IDEF for invalid syntax. These error exits are used in conjunction with the ASMERROR statement.

Coding $IDEF for the NEWCMD Instruction

In the following example, the instruction NEWCMD is defined with one positional (OP1) and two keyword (KWD and P1) operands. The error exits are at labels ERROR2 and ERROR3.

The $IDEF statement coded for NEWCMD in the overlay program looks like:

```
ASMOLAYX PROGRAM  BEGIN
         •
BEGIN    EQU      *
         •
NEWLIST  $IDEF    OP1,(KWD,P1),PERR=ERROR2,KERR=ERROR3
         •
ERROR2   •
ERROR3   •
```

Checking the Source Statement Syntax

When $EDXASM parses the NEWCMD instruction, it builds tables and pointers and stores this data in the compiler common area. $EDXASM passes the address of this area as a 1-word parameter. Your overlay program must refer to this parameter as $PARM1 and then move it to either software register #1 or #2. Using the ASMCOMM equates, you can then access the fields in the common area. You use these fields to check syntax and build object text.

To illustrate how $EDXASM parses an instruction, Figure 28 on page CU-88 shows an example of the parsed output if you coded the NEWCMD instruction as follows:

```
SAMPLE    NEWCMD     A,KWD=(4,#1),P1=X
```

An explanation of the parsing example follows Figure 28 .

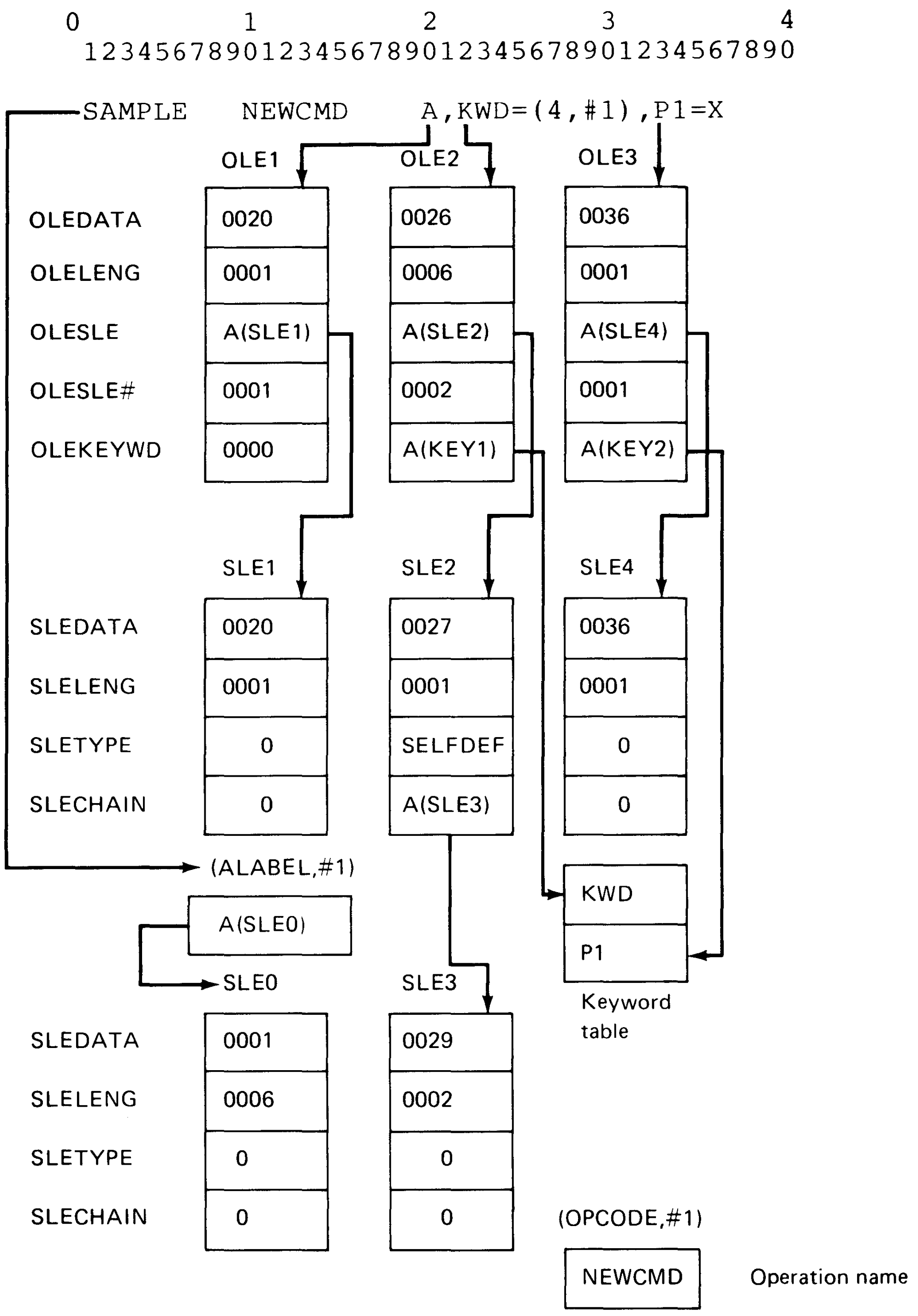


Figure 28. Source Statement Parsing Example

In this example, software register #1 points to the compiler common area, ASMCOMM. $EDXASM begins the parsing operation with the label SAMPLE and stores the results in the location (ALABEL,#1). $EDXASM creates a sublist element (SLE) for the label. A sublist element has four fields: SLEDATA, SLELENG, SLETYPE, and SLECHAIN. SLEDATA points to the first character of a label or operand. SLELENG is the number of characters in the label or operand. SLETYPE is the type of sublist element. SLECHAIN is used internally for creating chained sublist elements.

The SLETYPE field can have the value 0 (undefined), 1 (self-defining term), or 2 (string).

Self-defining terms are decimal constants (for example, 5, 1000, and -32000), hexadecimal constants (for example, X'1234', X'FF', and X'A0B0'), EBCDIC constants (for example, C'A' and C'12'), or symbols preceded by a + or - sign (for example, +LABEL1, +$DSCBLEN, and -LABEL2).

SLETYPE is "string" if the entire operand is enclosed in quotes. In this case, $EDXASM scans the entire data string for embedded double quotes which signify an apostrophe. If double quotes are found, $EDXASM changes them to single quotes and adjusts the SLE length field (SLELENG) accordingly.

In Figure 28 on page CU-88, the SLEDATA pointer for the label is 1, the field length is 6, and the type is undefined. If the source statement has no label, the compiler sets (ALABEL,#1) to 0.

$EDXASM enters the operation name (EDL instruction) in the field (OPCODE,#1). The compiler also generates a table of operand list elements that describe the coded operands. The word (AOPTABLE,#1) is the pointer to this table.

The table has a 10-byte header. Each operand list element (OLE) in the table is also 10-bytes in length. One OLE describes each operand.

An OLE has five fields: OLEDATA, OLELENG, OLESLE, OLESLE#, and OLEKEYWD. OLEDATA points to the first character of the operand. OLELENG is the number of characters in the operand. OLESLE points to the first sublist element (SLE) of the operand. The compiler generates at least one SLE for every operand. OLESLE# is the number of SLEs in the operand. If you coded a keyword operand, OLEKEYWD points to the keyword table that contains the the 1–7 character name of the keyword operand.

The sample NEWCMD source statement has three operands. The positional operand is A. The operand list element OLE1 describes this positional operand. The keyword operands are KWD= and P1=. These keyword operands are described by OLE2 and OLE3, respectively.

OLE1 indicates a 1-character operand at relative address 0020, with one SLE (SLE1). The operand type is undefined. OLE2 shows a 6-character operand beginning at 0026, with two SLEs (SLE2 and SLE3). SLE2 points to the constant 4 and SLE3 points to #1. OLE3 shows a 1-character operand at 0036, with one SLE (SLE4). SLE4 points to the X, whose type is undefined. $EDXASM stores the names of the keywords (KWD and P1) in the keyword table.

Creating an Overlay Program to Build the Instruction *(continued)*

The following code shows how to receive the address of the compiler common area and check for a valid instruction name. Control passes to label #NEWCMD upon a match; otherwise, control passes to label ERROR1.

```
ASMOLAYX PROGRAM  BEGIN,300,PARM=1
         COPY     ASMCOMM           COPY CODE FOR EQUATES
BEGIN    EQU      *
         MOVE     #1,$PARM1         GET ADDR OF COMMON AREA
         IF       ((OPCODE,#1),EQ,CNEWCMD,8),GOTO,#NEWCMD  CODE OK?
ERROR1   •
         •
         •
CNEWCMD  DC       CL8'NEWCMD'
#NEWCMD  EQU      *
```

You must now write the code to check syntax and handle syntax errors. You use the OPCHECK subroutine to check syntax against the model instruction. You use the ASMERROR statement to issue syntax error messages.

Using the sample overlay program, the code to check syntax and issue syntax error messages is shown:

```
ASMOLAYX PROGRAM  BEGIN,300,PARM=1
         COPY     ASMCOMM           COPY CODE FOR EQUATES
BEGIN    EQU      *
         MOVE     #1,$PARM1         GET ADDR OF COMMON AREA
         IF       ((OPCODE,#1),EQ,CNEWCMD,8),GOTO,#NEWCMD  CODE OK?
ERROR1   ASMERROR 1,$EDXLUSR        INVALID INSTRUCTION
ENDTASK  EQU      *                 SET UP EXIT
         DETACH
         GOTO     BEGIN
ERROR2   ASMERROR 2,$EDXLUSR        INVALID POSITIONAL OPERAND
ERROR3   ASMERROR 3,$EDXLUSR        INVALID KEYWORD
ERROR4   ASMERROR 4,$EDXLUSR        OPERAND ONE MISSING
         ASMERROR GENERATE
CNEWCMD  DC       CL8'NEWCMD'
NEWLIST  $IDEF    OP1,(KWD,P1),PERR=ERROR2,KERR=ERROR3     MODEL
         •
         •
         •
#NEWCMD  EQU      *
         CALL     OPCHECK,(NEWLIST)       CHECK SYNTAX
```

In the previous example, if the instruction name is not NEWCMD, you issue error message 1 (invalid instruction) and exit the program. To exit the program, you must code the label ENDTASK. ASMERROR statements branch to this label. In addition, you must end the overlay program with a DETACH followed by a GOTO to the first executable instruction in the overlay program. If the instruction name is NEWCMD, control passes to the label #NEWCMD.

Creating an Overlay Program to Build the Instruction *(continued)*

At label #NEWCMD, you call the OPCHECK subroutine. The OPCHECK subroutine compares the instruction syntax and fills in the tables and pointers of the compiler common area. Upon encountering syntax errors, control passes to the appropriate label you define on the $IDEF statement. In this example, ERROR2 and ERROR3 are the error exits.

Building Object Text

After OPCHECK executes, the tables and pointers in the compiler area contain the addresses of the operand list elements (OLEs) and sublist elements (SLEs). You use this data to build object text. The object text you build is called an object text element (OTE). You use the OTE statement to do this. $EDXASM uses OTEs to build object code for further processing.

Before you build OTEs, you must understand the format of the expanded object code. This is described next.

Expanded Object Code Format

The object code $EDXASM generates for NEWCMD will be either 2 or 3 words, depending on whether KWD is specified. This is illustrated in the next three examples. The label you code on NEWCMD is the label on the first word of the object code.

The first word is the operation code word and contains a flag byte (bits 0–7) and an operation code byte (bits 8–15). The operation code byte for NEWCMD contains a value of 1.

Figure 29 and Figure 30 on page CU-92 show the possible flag bit meanings for NEWCMD:

Bit	Meaning
0	This bit is on if operand 2 (KWD) is a constant
1	Keyword operand is specified (KWD)
2 & 3	Not used

Figure 29. Flag bit meanings (bits 0-3)

Creating an Overlay Program to Build the Instruction *(continued)*

Bits 4–7 indicate software register usage for operands 1 and 2 as follows:

Bits/operand	Register not used	#1 used as (x,#1)	#2 used as (x,#2)	#1 or #2 used as operand
4 & 5 for op2	00	01	10	11
6 & 7 for op1	00	01	10	11

Figure 30. Flag bit meanings (bits 4-7)

The second and third words are the address of the OP1 and KWD operands respectively. Both OP1 and KWD may be indexed and KWD may also be a self-defining term. If you code KWD, the object code is three words in length. Also, bit 1 of the operation code word is set to 1 (on). If you specify P1, P1 will be the label on the second word.

The next three examples show the expansion depending on how you code NEWCMD:

For example, if you code:

```
LABEL1    NEWCMD    X
```

$EDXASM generates the following object code:

```
LABEL1    DC        X'0001'    (bits 0-7 = 0000 0000)
          DC        A(X)
```

If you code:

```
LABEL2    NEWCMD    Y,KWD=Z,P1=AY
```

$EDXASM generates the following object code:

```
LABEL2    DC        X'4001'    (bits 0-7 = 0100 0000)
AY        DC        A(Y)
          DC        A(Z)
```

If you code:

```
LABEL3    NEWCMD   (4,#1),KWD=7,P1=OP1ADDR
```

$EDXASM generates the following object code:

```
LABEL3    DC        X'C101'    (bits 0-7 = 1100 0001)
OP1ADDR   DC        F'4'
          DC        F'7'
```

Defining the Object Text Elements

Upon completion of the OPCHECK subroutine, you must define and build the object text elements. The sample overlay program defines three OTEs for NEWCMD. The first OTE definition (NEWOTE1) builds an operation code OTE with a code of 1. You use the other two OTEs (NEWOTE2 and NEWOTE3) to build object text for the OP1 and KWD operands.

The following code defines the OTEs. In addition, the operation code and label from NEWCMD are placed in NEWOTE1:

```
         •
         •
         •
NEWLIST  $IDEF     OP1,(KWD,P1),PERR=ERROR2,KERR=ERROR3      MODEL
NEWOTE   DC        F'3'                    NUMBER OF OTES
NEWOTE1  OTE       TYPE=OPCODE,SLEDATA=1   SET OP CODE TO 1
NEWOTE2  OTE       TYPE=ADDRESS            OTE FOR "OP1"
NEWOTE3  OTE       TYPE=ADDRESS            OTE FOR "KWD"
#NEWCMD  EQU       *
         CALL      OPCHECK,(NEWLIST)       CHECK SYNTAX
*   INITIALIZE "OP CODE" OTE
         MOVE      NEWOTE1+OTEDATAP,1      RESET OP CODE TO 1
         MOVE      NEWOTE1+OTEDATAL,(ALABEL,#1)   INSERT LABEL
```

If a label does not exist on NEWCMD, (ALABEL,#1) is zero and $EDXASM does not generate a label. Note that although the operation code for NEWOTE1 is defined as 1 (SLEDATA=1), the operation code is reset to 1 on the MOVE instruction. Throughout your overlay program, you must reset any data fields that might change. This is because $EDXASM could invoke the program again without ever reloading it.

You must now process the operands of NEWCMD and build object text. The next section describes how you process the OP1 operand.

Processing the OP1 Operand

You process the OP1 operand first by storing the sublist element (SLE) for the P1 operand in the label field of NEWOTE2. This moves the address of the SLE which defines the label on P1= (if specified) into NEWOTE2. Processing the OP1 operand in this manner causes the label for operand 1 to be created.

Because NEWOTE2 is defined as an address OTE, you must store the sublist element (SLE) address that defines the label to be generated. In this case, OP1+2 contains the SLE address that defines the label.

Since OP1 is indexable, you must also indicate whether an index register is used for OP1. The flag bit settings in the operation code word indicate register usage. You use the $INDEX subroutine to store this information in the object text element for NEWOTE1.

Creating an Overlay Program to Build the Instruction *(continued)*

The following code processes OP1 and stores register usage information. If OP1 is missing, an error message is issued and the program exits:

```
         •
         •
         •
NEWOTE1  OTE      TYPE=OPCODE,SLEDATA=1     SET OP CODE TO 1
NEWOTE2  OTE      TYPE=ADDRESS              OTE FOR "OP1"
NEWOTE3  OTE      TYPE=ADDRESS              OTE FOR "KWD"
#NEWCMD  EQU      *
         CALL     OPCHECK,(NEWLIST)         CHECK SYNTAX
*    INITIALIZE "OP CODE" OTE
         MOVE     NEWOTE1+OTEDATAP,1        RESET OP CODE TO 1
         MOVE     NEWOTE1+OTEDATAL,(ALABEL,#1)    INSERT LABEL
*    PROCESS "OP1" OPERAND
         IF       (OP1+2,EQ,0),GOTO,ERROR4  OP1 MISSING?
         MOVE     NEWOTE2+OTEDATAL,P1+2     STORE ADDR OF P1 SLE
         MOVE     NEWOTE2+OTEDATAP,OP1+2    STORE ADDR OF OP1 SLE
         CALL     $INDEX,OP1,NEWOTE+OTEDATAP,(NEWOTE2),1
```

Now you must write the code to process the KWD operand. The next section describes how you do this.

Processing the KWD Operand

When you process the KWD operand, you must first determine whether it was coded on NEWCMD. If KWD is not coded, you must set the type field of NEWOTE3 to #NULL. This causes $EDXASM to ignore this OTE.

If KWD is coded, you must *reset* the type field of NEWOTE3 to #ADDRESS. Next, you must set flag bit 1 to 1 in the operation code word. This indicates that KWD is specified. You do this by IORing X'4000' into the operation code word.

Because NEWOTE3 is defined as an address OTE, you must store the sublist element (SLE) address that defines the data to be generated. In this case, KWD+2 contains the SLE address which defines the data.

Similar to the OP1 operand, KWD is also indexable. Again, you use the $INDEX subroutine to store the appropriate bits in NEWOTE1.

Creating an Overlay Program to Build the Instruction *(continued)*

The code you use to process the KWD operand follows:

```
*    PROCESS "KWD" OPERAND
          IF        (KWD,EQ,0)                    KWD SPECIFIED?
            MOVE    NEWOTE3+OTETYPE,+#NULL        SET OTE TYPE TO NULL
          ELSE
            MOVE    NEWOTE3+OTETYPE,+#ADDRESS     RESET TYPE TO ADDRESS
            IOR     NEWOTE1+OTEDATAP,X'4000'      SET FLAG BIT 1 ON
            MOVE    NEWOTE3+OTEDATAP,KWD+2        STORE ADDR OF KWD
            CALL    $INDEX,KWD,NEWOTE1+OTEDATAP,(NEWOTE3),2
          ENDIF
```

You must now write the code to exit the overlay program and return control back to $EDXASM. This is described next.

Ending the Overlay Program

After you process all the operands, you must store the number of OTEs built in the overlay program. You do this by passing the address of the OTE count word, in this case NEWOTE. You must then issue a GOTO to the label ENDTASK. $EDXASM generates the object code for NEWCMD when the ENDTASK exit is taken.

The code you use to exit the overlay program follows:

```
*    SET UP EXIT
          MOVEA     (AMACDATA,#1),NEWOTE          STORE OTE COUNT
          GOTO      ENDTASK
          COPY      COPCHECK              COPY CODE FOR OPCHECK SUBRTN
          COPY      C$INDEX               COPY CODE FOR $INDEX SUBRTN
          ENDPROG
          END
```

Creating an Overlay Program to Build the Instruction *(continued)*

Sample Overlay Program for NEWCMD

The coding segments throughout this section showed you how to create an overlay program. The following is the overlay program in its entirety:

```
ASMOLAYX PROGRAM  BEGIN,300,PARM=1
         COPY     ASMCOMM            COPY CODE FOR EQUATES
BEGIN    EQU      *
         MOVE     #1,$PARM1          GET ADDR OF COMMON AREA
         IF       ((OPCODE,#1),EQ,CNEWCMD,8),GOTO,#NEWCMD  CODE OK?
ERROR1   ASMERROR 1,$EDXLUSR         INVALID INSTRUCTION
ENDTASK  EQU      *                  SET UP EXIT
         DETACH
         GOTO     BEGIN
ERROR2   ASMERROR 2,$EDXLUSR         INVALID POSITIONAL OPERAND
ERROR3   ASMERROR 3,$EDXLUSR         INVALID KEYWORD
ERROR4   ASMERROR 4,$EDXLUSR         OPERAND ONE MISSING
         ASMERROR GENERATE
CNEWCMD  DC       CL8'NEWCMD'
NEWLIST  $IDEF    OP1,(KWD,P1),PERR=ERROR2,KERR=ERROR3     MODEL
NEWOTE   DC       F'3'                     NUMBER OF OTES
NEWOTE1  OTE      TYPE=OPCODE,SLEDATA=1    SET OP CODE TO 1
NEWOTE2  OTE      TYPE=ADDRESS             OTE FOR "OP1"
NEWOTE3  OTE      TYPE=ADDRESS             OTE FOR "KWD"
#NEWCMD  EQU      *
         CALL     OPCHECK,(NEWLIST)        CHECK SYNTAX
*    INITIALIZE "OP CODE" OTE
         MOVE     NEWOTE1+OTEDATAP,1       RESET OP CODE TO 1
         MOVE     NEWOTE1+OTEDATAL,(ALABEL,#1)    INSERT LABEL
*    PROCESS "OP1" OPERAND
         IF       (OP1+2,EQ,0),GOTO,ERROR4  OP1 MISSING?
         MOVE     NEWOTE2+OTEDATAL,P1+2     STORE ADDR OF P1 SLE
         MOVE     NEWOTE2+OTEDATAP,OP1+2    STORE ADDR OF OP1 SLE
         CALL     $INDEX,OP1,NEWOTE+OTEDATAP,(NEWOTE2),1
*    PROCESS "KWD" OPERAND
         IF       (KWD,EQ,0)                KWD SPECIFIED?
           MOVE   NEWOTE3+OTETYPE,+#NULL    SET OTE TYPE TO NULL
         ELSE
           MOVE   NEWOTE3+OTETYPE,+#ADDRESS  RESET TYPE TO ADDRESS
           IOR    NEWOTE1+OTEDATAP,X'4000'   SET FLAG BIT 1 ON
           MOVE   NEWOTE3+OTEDATAP,KWD+2     STORE ADDR OF KWD
           CALL   $INDEX,KWD,NEWOTE1+OTEDATAP,(NEWOTE3),2
         ENDIF
*    SET UP EXIT
         MOVEA    (AMACDATA,#1),NEWOTE       STORE OTE COUNT
         GOTO     ENDTASK
         COPY     COPCHECK            COPY CODE FOR OPCHECK SUBRTN
         COPY     C$INDEX             COPY CODE FOR $INDEX SUBRTN
         ENDPROG
         END
```

Figure 31. Sample overlay program

Creating an Overlay Program to Build the Instruction *(continued)*

After you complete the coding of the overlay program, you must compile it using $EDXASM. You must create a load module by using either $UPDATE or $EDXLINK. You must specify the name of the load module in a language control data set extension. How and why you do this is described in the section "Creating a Language Control Data Set Extension."

Creating a Language Control Data Set Extension

$EDXASM uses a language control data set to generate syntax error messages and to locate overlay programs. The $EDXL data set contains this information. You create an extension to $EDXL to contain your error messages and overlays. Creating an extension to $EDXL minimizes the changes you would have to make if you receive a new version of $EDXL or $EDXASM.

A language control data set is divided into two logical parts. The first part contains the syntax error messages. The second part contains the names of overlay programs and instructions. Each overlay has a corresponding instruction which it processes. The second part also contains the names of the copy code modules that you might reference in an assembly. The extension you create has this same format.

There are five control statements you can use in a language control data set. The following is a brief description of these control statements:

***COMMENT**	Indicates a comment
***COPYCOD**	Defines a copy code library
***EXTLIB**	Defines a language control data set extension
***OVERLAY**	Defines an overlay program and the instruction(s) it processes
****STOP****	Indicates the end of a language control data set

The format and description of the control statements are in the section "Control Statements" on page CU-99.

This section shows how to create an extension for the NEWCMD instruction. You use a text editor to create the extension.

Entering the Syntax Error Messages

In the sample overlay program, four syntax messages were defined. The ASMERROR statement was used to indicate the message number (1–4). The messages you enter in this data set and their line numbers must correspond to the ASMERROR message numbers.

Creating a Language Control Data Set Extension *(continued)*

You begin the message in column 2. The numbers you enter in columns 2 and 3 indicate the completion code. $EDXASM does not generate object code if you specify a completion code greater than 8.

The messages for NEWCMD look like the following:

```
 08 *** INVALID OR UNDEFINED OPERATION CODE
 08 *** AN INVALID POSITIONAL OPERAND WAS SPECIFIED
 08 *** AN INVALID KEYWORD PARAMETER WAS SPECIFIED
 08 *** OPERAND ONE IS MISSING
```

Following the syntax messages, you must specify the overlay program and instruction names.

Specifying the Overlay and Instruction Names

You use the *OVERLAY statement to define the name of the overlay program, the volume it resides on, and the instruction(s) the overlay processes. This statement must begin in column 1.

Assuming the load module for the sample overlay program is in data set NEWOLAY on volume ASMLIB, the *OVERLAY statement would look like:

```
 08 *** INVALID OR UNDEFINED OPERATION CODE
 08 *** AN INVALID POSITIONAL OPERAND WAS SPECIFIED
 08 *** AN INVALID KEYWORD PARAMETER WAS SPECIFIED
 08 *** OPERAND ONE IS MISSING
*OVERLAY NEWOLAY  ASMLIB   NEWCMD
```

You must enter a statement to indicate the end of the extension data set. You enter the **STOP** statement beginning in column 1 to do this. The complete extension data set now looks like:

```
 08 *** INVALID OR UNDEFINED OPERATION CODE
 08 *** AN INVALID POSITIONAL OPERAND WAS SPECIFIED
 08 *** AN INVALID KEYWORD PARAMETER WAS SPECIFIED
 08 *** OPERAND ONE IS MISSING
*OVERLAY NEWOLAY  ASMLIB   NEWCMD
**STOP**
```

Because the name $EDXLUSR is specified on the ASMERROR statements in the overlay program, you must save the extension with that name.

After you save the language control data set extension, you must specify its name and volume in $EDXL. You do this by editing $EDXL and entering an *EXTLIB statement beginning in column 1.

An example of what $EDXL would look like with the *EXTLIB statement follows:

```
08 *** TOO MANY POSITIONAL OPERANDS WERE SPECIFIED
08 *** AN INVALID KEYWORD PARAMETER WAS SPECIFIED
08 *** ONE OR MORE UNDEFINED LABELS WERE REFERENCED
    •
    •
    •
08 *** PART NOT ALLOWED WITH DSX SPECIFICATIONS
*OVERLAY $ASM0008 ASMLIB  MOVE  MOVEA  AND  IOR  EOR
*OVERLAY $ASM0009 ASMLIB  WAIT  POST   ENQ  DEQ
*COMMENT
*OVERLAY $ASM0003 ASMLIB   PROGRAM  LOAD     DSCB
*EXTLIB  $EDXLUSR ASMLIB
*COPYCOD ASMLIB
*COPYCOD EDX002
**STOP**
```

After you create the language control data set extension and update $EDXL, the next step is to add the operation code for NEWCMD. The procedure for doing this is described in "Defining the Instruction Operation Code" on page CU-101.

Control Statements

This section describes the control statements you can use in a language control data set.

*OVERLAY Statement

You use the *OVERLAY statement to define the name of the overlay program, the volume that it resides on, and the instructions that it processes.

The *OVERLAY statement has the following format:

Column	Contents
1–8	*OVERLAY
10–17	Program name
19–24	Volume name (blank indicates IPL volume)
28–35	Instruction 1
37–44	Instruction 2
46–53	Instruction 3
55–62	Instruction 4
64–71	Instruction 5.

Creating a Language Control Data Set Extension *(continued)*

If an overlay program processes more than five instructions, you continue the instruction names in column 1 on the next line. You can specify up to eight instruction names on the continued line. Each instruction is allowed eight columns and one blank. Instructions would begin, for example, in columns 1, 10, and 19.

*EXTLIB Statement

You use the *EXTLIB statement to define a language control data set extension. This data set contains additional error message text, overlay and instruction names, and copy code volume names. The extension data set has the same format and characteristics as the primary language control data set ($EDXL).

The *EXTLIB statement has the following format:

Column	Contents
1–7	*EXTLIB
10–17	Language extension data set name
19–24	Volume name (blank indicates the IPL volume).

You should always insert this statement *before* any *COPYCOD statements in the primary language control data set.

*COPYCOD Statement

You use the *COPYCOD statement to define a copy code library. The volume you specify contains source code modules you reference on the compiler COPY statement.

The *COPYCOD statement has the following format:

Column	Contents
1–8	*COPYCOD
10–17	Volume name.

The language control data set or its extensions may contain up to five different *COPYCOD statements. When $EDXASM processes compiler COPY statements, it searches the defined *COPYCOD volumes in the order in which the *COPYCOD statements occur in the language control data set.

*COMMENT Statement

You use the *COMMENT statement to insert optional comments in the language control data set. $EDXASM ignores the text you specify on this statement.

STOP Statement

You use the **STOP** statement to indicate the end of the language control data set. You can add additional error messages, overlay programs, and copy code modules after this point. The number of additional modules is limited by the size of the operation code table (OPCTABLE).

Defining the Instruction Operation Code

Every EDL instruction must have its operation code defined in the emulator command table. This section explains how you can define the operation code for NEWCMD through the use of an initialization routine. This routine will execute every time you IPL.

The following code inserts the operation code into the emulator command table. An explanation of this routine follows the example.

```
         PROGRAM   MAIN=NO
         COPY      PROGEQU              PROGRAM HDR EQUATES
         EXTRN     INITEXIT,MYRTN       DEFINE EXTERNAL ENTRY PTS
         ENTRY     CMDINIT
CMDINIT  EQU       *
         MOVE      #1,$CMDTABL          EMULATOR COMMAND TABLE
         MOVEA     (2,#1),MYRTN         DEFINE OP CODE 01 PROCESS RTN
         GOTO      INITEXIT             BRANCH TO SUPV INIT RTN
         ENDPROG
         END
```

The routine includes the PROGEQU equates. Doing this resolves references to $CMDTABL. $CMDTABL contains the addresses of the routines that do the processing for EDL instructions. Next, the routine defines two external entry points: INITEXIT and MYRTN. INITEXIT is an entry point in the supervisor to which your routine must return control upon exit. MYRTN is the entry point of the Series/1 assembler program that processes the NEWCMD instruction. This routine is described later.

The code beginning at entry point CMDINIT places the address of MYRTN in the emulator command table. The MOVE instruction moves the starting address of the emulator command table ($CMDTABL) into software register 1. The MOVEA instruction moves the address of MYRTN two bytes into the table. Hence, when the emulator encounters operation code 01, the emulator passes control to MYRTN.

Note: Operation codes 01 and 02 are reserved for your use. To define operation code 02, move the address of the routine four bytes into the emulator command table.

You exit the routine by branching to label INITEXIT.

You must assemble and link-edit this routine with the supervisor. You specify the entry point name CMDINIT on the INITMOD= operand of the SYSTEM statement at system generation.

The entry point MYRTN, defined as an external, must be the entry point of the routine that processes NEWCMD. The Series/1 assembler code required for this routine is described next.

Creating Your Own EDL Instruction

Writing the Assembler Code for NEWCMD

This section shows the Series/1 assembler code that performs the function of NEWCMD. For the instruction you create, you must also write the Series/1 assembler code that performs the function you need. Refer to the *IBM Series/1 Event Driven Executive Macro Assembler*, GC34-0317 for details on how to code in Series/1 assembler.

You will need the Series/1 Macro Assembler ($S1ASM) to perform this step.

Coding Considerations

When you code your Series/1 assembler routine, adhere to the following:

- Write the routine in Series/1 assembler code only.
- Follow the register conventions used by CMDSETUP.
- Ensure the routine is reentrant:
  - no subroutines are used.
  - no parameter naming operands (Px=) are coded.
  - data areas are unique to each task.
  - always test R5 for the operation code.
  - ensure R2 contains the TCB address upon exit.
  - ensure R1 is incremented by the instruction length (in bytes) upon exit.

Writing the Assembler Code for NEWCMD *(continued)*

Description of Sample Program

Again, if you code one operand on the NEWCMD instruction, it adds 1 to the value of operand 1. If you code two operands, the value of operand 2 is added to the value of operand 1. The following description explains how this is done:

At the entry point MYRTN, the routine begins by checking the flag bits of the operation code in register 5 (R5). The flag bits indicate whether one or two operands were specified. If bit 1 equals 1, only one operand was coded on NEWCMD. The routine branches to label OPND1 to process operand 1. Here, the routine adds the value 1 to the value of operand 1 (R3). Next, register 1 (R1) is incremented by the length of NEWCMD with one operand coded. In this case, NEWCMD is four bytes in length. After R1 is incremented, the routine branches to CMDSETUP. CMDSETUP then processes the next source statement in the source program. The function of CMDSETUP is described in detail in the *Internal Design*.

The code at label OPND2 is executed when bit 1 of the operation code equals 0. This bit indicates both operand 1 and operand 2 were coded. The value of operand 2 (R4) is added to the value of operand 1 (R3). Next, register 1 (R1) is incremented by six bytes. After R1 is incremented, the routine branches to CMDSETUP.

```
         ENTRY   MYRTN
MYRTN    EQU     *
*
*  CMDSETUP REGISTER CONVENTIONS:
*        R1 ==>  OP CODE
*        R2 ==>  TCB
*        R3 ==>  OP1 ADDRESS
*        R4 ==>  OP2 ADDRESS OR DATA (IF IT EXISTS)
*        R5 ==>  OP CODE
*
*
CHKBITS  TWI     X'4000',R5          TEST IF BIT 1 OFF; IF OFF
*                                    THERE IS ONLY ONE OPERAND
         JOFF    OPND1               LABEL FOR ONLY ONE OPERAND
OPND2    AW      (R4),(R3)           ADD OP2 TO OP1
         AWI     6,R1                SET UP R1 FOR NEXT INSTRUCTION
         BX      CMDSETUP            BRANCH BACK TO EMULATOR
OPND1    EQU     *
         AWI     1,(R3)              ADD 1 TO OP1
         AWI     4,R1                SET UP R1 FOR NEXT INSTRUCTION
         BX      CMDSETUP            BRANCH BACK TO EMULATOR
         END
```

Use $S1ASM to assemble this routine. You must link-edit the assembled output from this routine and the output from the initialization routine with the supervisor.

Testing the New Instruction

You should write a small program containing the new instruction to test it. Testing the new instruction will indicate if the overlay program, initialization routine, and assembler routine work properly.

System Generation Requirements

Before you test the instruction, make sure you do the following:

1. Use a text editor to read in the link-control data set that defines the modules currently in your supervisor (normally LINKCNTL on EDX002).
2. Specify INCLUDE statements for the assembled output from the initialization routine and the assembler routine. You specify the names of these data sets.
3. Write (save) the updated link-control data set back to LINKCNTL.
4. Use a text editor to read in the data set that defines your current system configuration (normally $EDXDEFS on EDX002).
5. Code the INITMOD= operand on the SYSTEM statement. You must specify the entry point name of your initialization routine. For the NEWCMD instruction, specify the entry point name CMDINIT.
6. Write (save) the updated data set back to $EDXDEFS.
7. Perform a system generation.
8. After the system generation completes, initialize (II command of $INITDSK) the new supervisor and IPL the system.

When you complete these steps, you can test your instruction.

Coding a Test Program

When you test the instruction, you should code all the possible variations of the instruction's syntax. You should also test for invalid syntax.

You can use the following sample program to test the NEWCMD instruction:

```
TEST      PROGRAM   BEGIN
BEGIN     EQU       *
          NEWCMD    A                      ADD 1 TO A (1)
          NEWCMD    A,KWD=B                ADD B (2) TO A (2)
          PRINTEXT  '@THE RESULT IS: '
          PRINTNUM  A                      A = 4
          MOVEA     #1,VALUES              SET UP INDEX
          NEWCMD    C,KWD=(4,#1)           ADD D (5) TO C (3)
          PRINTEXT  '@THE RESULT IS: '
          PRINTNUM  C                      C = 8
          MOVEA     AY,X                   SET ADDR OF X
          NEWCMD    A,P1=AY                USE X AND ADD 1
          PRINTEXT  '@THE RESULT IS: '
          PRINTNUM  X                      X = 1
*
* INVALID SYNTAX - THESE GENERATE ERROR MESSAGES
*
          NEWCMD     X,KWDD
          NEWCMD     X,P2=ERR
*
*
          PROGSTOP
A         DATA       F'1'
VALUES    EQU        *
B         DATA       F'2'
C         DATA       F'3'
D         DATA       F'5'
X         DATA       F'0'
          ENDPROG
          END
```

If the overlay program is correct, the compiler listing for the test program will show the object code generated for the valid statements. Further, $EDXASM should issue error messages for the statements with invalid syntax.

Upon receiving a -1 completion code from $EDXASM, create a load module using $UPDATE or $EDXLINK. Load the program via the $L command to execute the program. The output from your program should yield the expected results.

Debugging Overlay Programs

You can use $DEBUG to debug an overlay program. To do this, you must:

1. Code a READTEXT, QUESTION, or WAIT KEY instruction as the first executable instruction of the overlay program. When the overlay program is loaded, it will stop at this instruction and wait for input from the terminal.

2. Load $EDXASM and specify one overlay area (OV option) when you compile the source program containing your new EDL instruction.

3. Load $DEBUG in the same partition as $EDXASM when $EDXASM loads your overlay program and the overlay program stops at the READTEXT, QUESTION, or WAIT KEY.

4. Enter $ASMOPCD when $DEBUG prompts you for the program name. If $ASMOPCD is already in storage, do not request a new copy to be loaded.

Once the overlay program is in storage, you can examine data areas and set breakpoints with $DEBUG.

If a program check occurs in the overlay program, the system cancels the overlay program and issues a program check message. The error message may not give the correct displacement into the overlay program for the failing instruction (R1) and the TCB address (R2). If these addresses appear to be outside the program, you can calculate the correct addresses by subtracting the program load point address from the address of R1 and R2. The resulting addresses may be in either $EDXASM or in one of the overlay programs.

Creating Unique Labels Within the Overlay Program

Instructions may require unique labels which do not conflict with labels you create from a previous call to your overlay program or labels define in an application program. For example, $EDXASM creates a unique label (internally) for each ENDIF statement when multiple IF-ENDIF statements are coded in a program.

$EDXASM provides a method for you to create unique labels when you use the field $SYSNDX in an overlay program. $SYSNDX is a 1-word field in the compiler common area. You reference this field through the ASMCOMM equates.

$EDXASM sets up a 4-digit counter for this field. You must add 1 to this counter to generate a unique label each time you use $SYSNDX. You can convert the binary value of $SYSNDX to a 4-character EBCDIC representation of the number using the CONVTB instruction. The following example shows how to convert the value of $SYSNDX. Assume that #1 points to the compiler common area and that $SYSNDX contains the value 2. After the conversion, INDEX contains the character value "0002".

```
         CONVTB    INDEX,($SYSNDX,#1),FORMAT=(4,0,I)
         •
         •
         •
INDEX    DC        CL4'0000'
```

After conversion, you append the four characters to a 1- to 4- character prefix to form a unique label. For example, the following code shows how to define a unique label with the prefix $$LI using the value in INDEX from the previous example:

```
         •
         •
         •
         MOVE      LAB1+4,INDEX,(4,BYTES)
         ADD       ($SYSNDX,#1),1
         •
         •
         •
OTE1     OTE       TYPE=ADDRESS,SLENAME=SLE1
SLE1     SLE       ADDRESS=LAB1,LENGTH=8
LAB1     DC        CL8'$$LI'
INDEX    DC        CL4'0000'
```

The name on the label created would have the text $$LI0002. You could then refer to this label in other object text elements.

Generating Source Statements

An overlay program can generate one source statement which $EDXASM processes after the generating overlay ends. $EDXASM processes this source statement before processing the next statement in the source data set. One instance where this feature is used, is when you specify TASK=YES on a DISK statement. The overlay program $ASM000S, which processes the DISK statement, creates a TASK statement for the device's disk task. The overlay program $ASM000T, which processes the TERMINAL statement, also uses this feature to generate keyboard tasks for terminals.

You can use this feature in your overlay program to generate a source statement and optionally create a continuation line for that statement.

Notes:

1. If you built an instruction in the overlay program, the source statement must also be an instruction. If you built a statement in the overlay program, the source statement must also be a statement (nonexecutable).

2. The source statement you create does not appear in the compiler listing; however, the object code generated does appear if the source statement is an instruction.

3. If a compilation error occurs with the source statement you create, the error message appears after the instruction or statement you built in the overlay program.

Generating Source Statements *(continued)*

Creating a Source Statement — No Continuation Line

To create a source statement (with no continuation line), do the following:

1. Define an 80-byte area in the overlay program which contains the text of the source statement. For example, the following statement:

   ```
   TRMNL     IOCB      SCREEN=ROLL
   ```

   looks as follows when defined in the overlay program:

   ```
   SRCSTMT   DATA      CL80'TRMNL     IOCB      SCREEN=ROLL'
   ```

2. Move the field #AINBUF to a software register. This field is defined in ASMCOMM and contains an address of a storage area. The address to which #AINBUF is pointing is where you move the source statement. For example, if #1 points to ASMCOMM, the following code shows what you must do to move the source statement to #2:

   ```
             MOVE      #2,(#AINBUF,#1)          GET ADDR OF STORAGE AREA
             MOVE      (0,#2),SRCSTMT,(80,BYTES)   MOVE SOURCE STATEMENT
   ```

3. Move the value X'FFFF' to #AINBUF. This move indicates that the overlay program is creating a source statement and that $EDXASM must process it before the next statement in the source data set. After moving X'FFFF' to #AINBUF, store the number of object text elements created in the overlay, and return control to label ENDTASK to exit the overlay program. The following example shows how to set #AINBUF and exit the overlay program:

   ```
             MOVE      (#AINBUF,#1),X'FFFF'     GENERATE SOURCE FLAG
             MOVEA     (AMACDATA,#1),NEWOTE     PASS OTE COUNT
             GOTO      ENDTASK
   ```

Generating Source Statements *(continued)*

Creating a Source Statement — With Continuation Line

To create a source statement with a continuation line, you do the same steps previously discussed plus some additional steps. These additional steps are explained in this section.

After you define the source statement within the 80-byte area, do the following:

1. Insert a nonblank character in column 72 of the source statement.

2. Move the address, to column 77 of the source statement, of a subroutine that will append the continuation line to the source statement.

The subroutine, which you must write and define in the overlay program, must be written to receive the address of a storage area. $EDXASM calls the subroutine (*after* the overlay program branches to ENDTASK) and passes the subroutine an address. Because $EDXASM defines the storage area for you, do not define this area in the overlay program. The subroutine must use the buffer area at that address to construct the continued source statement.

The following is an example of how you can do this:

```
          •
          •
          •
          MOVE      #2,(#AINBUF,#1)             GET ADDR OF STORAGE AREA
          MOVE      (0,#2),SRCSTMT,(80,BYTES)   MOVE SOURCE STATEMENT
          MOVE      (71,#2),C'X',(1,BYTE)       SET CONTINUATION FLAG
          MOVEA     (76,#2),CONTSUB             MOVE ADDR OF SUBRTN
          MOVE      (#AINBUF,#1),X'FFFF'        GENERATE SOURCE FLAG
          MOVEA     (AMACDATA,#1),NEWOTE        PASS OTE COUNT
          GOTO      ENDTASK
SRCSTMT   DATA      CL80'TRMNL    IOCB      SCREEN=ROLL,'
CONTSRC   DATA      CL80'PAGSIZE=60,NHIST=6'
CONTSAVE  DATA      F'0'                        SAVE AREA ADDR
          SUBROUT   CONTSUB,CONTBUF
          MOVE      CONTSAVE,#2                 SAVE CONTENTS OF #2
          MOVE      #2,CONTBUF                  GET BUFFER ADDR
          MOVE      (0,#2),C' ',(80,BYTES)      SET BUFFER TO BLANKS
          MOVE      (15,#2),CONTSRC,(18,BYTES)  BUILD NEXT LINE
          MOVE      (71,#2),C' ',(1,BYTE)       CLEAR CONTINUE COLUMN
          MOVE      #2,CONTSAVE                 RESTORE #2
          RETURN
```

The source statement created in the previous example and passed to $EDXASM looks like the following:

```
TRMNL     IOCB      SCREEN=ROLL,                                          X
                 PAGSIZE=60,NHIST=6
```

Overlay Program Statements

This section describes in detail the overlay program statements you can use and their coding syntax.

$IDEF Statement — Build Model EDL Instruction

You use the $IDEF statement to build a model of the instruction. When you code $IDEF, you specify the positional operands and keywords of the instruction. The number of positional and keyword operands for an instruction must not exceed 50.

You can optionally specify error exits on $IDEF for invalid syntax. These error exits are used in conjunction with the ASMERROR statement.

The following is the syntax for the $IDEF statement:

Syntax:

```
label        $IDEF     posits,kwds,PERR=,KERR=
Required:    none
Defaults:    PERR=INVALPOS,KERR=INVALKWD
Indexable:   none
```

Operand	***Description***
posits	The list of allowable positional operands.
kwds	The list of allowable keyword operands. The keywords can be 1–7 characters in length. The keyword you specify is the actual keyword coded for the new instruction.
PERR=	The label of an instruction to branch to if more positional operands are coded in the instruction than defined by the instruction model. If omitted, control is passed to label INVALPOS, which you must code.
KERR=	The label of an instruction to branch to if a keyword operand is coded in the instruction which is not listed in the instruction model. If omitted, control is passed to label INVALKWD, which you must code.

Examples of $IDEF

The following are examples of how to code the $IDEF statement:

```
MODEL1    $IDEF     (POS1,POS2),KWD
MODEL2    $IDEF     POS,(MODE,LINE,SKIP,SPACES)
MODEL3    $IDEF     POS,KWD,PERR=BADPOS,KERR=BADKWD
```

Overlay Program Statements *(continued)*

ASMERROR Statement — Generate Syntax Error Messages

The ASMERROR statement generates a syntax error message for the input statement currently being processed if you code that statement incorrectly. ASMERROR is used in conjunction with the $IDEF statement. $EDXASM passes control to the label ENDTASK after the message is issued.

Note: A control block is required in the overlay program for you to use ASMERROR statement. You create the control block by coding:

```
ASMERROR GENERATE
```

You code ASMERROR GENERATE only once in a program.

Syntax:

```
label         ASMERROR number,extlib,P1=
Required:     number
Defaults:     extlib - $EDXL
Indexable:    none
```

Operand	***Description***
number	Code a decimal number representing the error message number to be generated. This number corresponds to a line number in the language control data set ($EDXL or extension). If this number is greater than the maximum error text line number, a general error message is printed.
extlib	The data set $EDXL or the name of the language control data set extension in which the error message text is located. This name must correspond to the data set name on an *EXTLIB control entry when you invoke $EDXASM. If the specified data set is not included as an extension to the primary language control data set ($EDXL), a general error message with asterisks for the data set name is printed. This data set is not used for an error message in the primary language control data set.

Examples of ASMERROR

The following are examples of the ASMERROR statement:

```
INVALPOS  ASMERROR  1
INVALKWD  ASMERROR  2
          ASMERROR  17,$EDXLUSR
```

Note: You can use the first two examples for the default error exits on the $IDEF statement. Messages 1 and 2 produce messages appropriate to these errors.

OTE Statement — Build Object Text Element

The OTE statement defines an object text element. You can use an object text element to do the following:

- Define a label
- Generate one or more bytes of object code
- Generate error messages
- Define external references and entry points.

The compiler aligns the object code on an even-byte address for TYPE=OPCODE, ADDRESS, and FCON.

Syntax:

```
label          OTE      TYPE=,DUPFAC=,SLEDATA=,SLENAME=
Required:      TYPE=
Defaults:      DUPFAC=1,SLEDATA=0,SLENAME=0
Indexable:     none
```

Operand	*Description*
TYPE=	The type of object text element to be defined. The ASMCOMM equate field OTETYPE defines this operand. The following types are valid:

NULL	OTE is to be ignored.
OPCODE	Data is an operation code. The SLEDATA operand contains the 2-byte operation code.
ADDRESS	Data is an address. The SLEDATA operand must point to the sublist element (SLE) defining the address constant.
ERROR	Generate an error message. The SLEDATA operand defines the numerical error message to be printed. This number corresponds to a line number in the primary language control data set ($EDXL).
FCON	Data is a fullword constant. The SLEDATA operand contains the two bytes of data to be generated.

Overlay Program Statements *(continued)*

DATA	Define untranslated data. The SLEDATA operand must point to a sublist element defining the data.
EQUATE	A label at the current location counter (for example LOC1 EQU *). The SLENAME operand must point to the SLE of the label. The SLEDATA operand should point to an SLE which points to the asterisk. Note that if you require an equate for other purposes, you can use the LABELS subroutine.
EXTRN	An external reference. The SLEDATA operand points to the SLE defining the name of the external symbol.
WXTRN	A weak external reference. The SLEDATA operand points to the SLE defining the name of the external symbol.
ENTRY	An entry point. The SLEDATA operand points to the SLE defining the symbol which is to be an entry point.

DUPFAC= Specifies the duplication factor for the object text element, or the number of times $EDXASM is to duplicate the object text in the object file. Only the first byte of text has the label defined by SLENAME.

You use this operand primarily for duplicating data definition fields, for example 128F‘0’.

If you specify DUPFAC=0, $EDXASM does not generate object text, but does do boundary alignment. This is equivalent to coding:

```
         ALIGN    WORD
```

The ASMCOMM equate field OTEDATAC defines this operand.

SLEDATA= If TYPE=OPCODE or FCON, SLEDATA defines the data to be entered into the object file. If TYPE=ERROR, it defines the error message number to be printed. If TYPE=ADDRESS, DATA, EXTRN, WXTRN, or ENTRY, it must contain the address of the sublist element (SLE) defining the data to be processed.

The ASMCOMM equate field OTEDATAP defines this operand.

SLENAME= The label assigned to the first byte of object text generated by the current OTE. If this field contains a 0, no label is assigned. Otherwise, it must contain the address of the SLE defining the label to be defined.

The ASMCOMM equate field OTEDATAL defines this operand.

Examples of OTE

The following are examples of the OTE statement:

```
OTE1     OTE       TYPE=ADDRESS
         OTE       TYPE=FCON,SLEDATA=0
         OTE       TYPE=EXTRN,SLEDATA=SLE1
```

Overlay Program Statements *(continued)*

SLE Statement — Build Sublist Element

The SLE statement enables you to define a sublist element in the same format as a sublist element generated by $EDXASM. You must use the SLE statement to generate a label or a data string that does not appear in the original input data.

Syntax:

```
label        SLE       ADDRESS=,LENGTH=,TYPE=
Required:    ADDRESS=,LENGTH=
Defaults:    TYPE=0 (address)
Indexable:   none
```

Operand	***Description***
ADDRESS=	The address of the text string defining the data. The ASMCOMM equate field SLEDATA defines this operand.
LENGTH=	The number of characters in the text string. The ASMCOMM equate field SLELENG defines this operand.
TYPE=	Omit this operand if the data defines an address; otherwise, specify either SELFDEF or STRING. The ASMCOMM equate field SLELENG defines this operand.

SELFDEF	Specify a self-defining term (for example, decimal or hexadecimal constants).
STRING	Specify string data. You must process this data by coding an OTE with TYPE=DATA specified.

Examples of SLE

The following are examples of the SLE statement:

```
SLE1     SLE      ADDRESS=NAME1,LENGTH=3
SLE2     SLE      ADDRESS=NAME2,LENGTH=1,TYPE=SELFDEF
SLE3     SLE      ADDRESS=ASTERISK,LENGTH=1

NAME1    DC       CL3'XYZ'    label XYZ
NAME2    DC       CL1'5'      constant 5
ASTERISK DC       CL1'*'      current location counter
```

Overlay Program Subroutines

This section describes in detail the overlay program subroutines you can use and their coding syntax.

$INDEX Subroutine -- Indicate Index Register Usage

The $INDEX subroutine examines an operand field for index register specification. It also stores control information in the operation code word and in the object text element for the operand being processed.

The $INDEX subroutine is in the form of copy code. You must include a COPY C$INDEX statement in your program to use it.

The CALL to the $INDEX subroutine has the following syntax:

Syntax:

```
[illegible]
```

Operand	*Description*
$INDEX	Code $INDEX as the first operand on the CALL instruction.
ole	The address of the operand list element (OLE) of the operand being processed.
opword	The address of the operation code word into which index register usage indicators may be set.
ote	The address of the object text element (OTE) that indicates the type of input.
posit	The position number (1, 2, or 3) of the input operand on the source statement.

[illegible]

You must store the SLE address of the operand being processed in the appropriate OTE before you call $INDEX.

[illegible]

How the "ole" operand is presented to $INDEX determines how the register flag bits are set in "opword." The flag bit settings are shown in Figure 32.

Overlay Program Subroutines *(continued)*

Bits/operand	Register not used	#1 used as (x,#1)	#2 used as (x,#2)	#1 or #2 used as operand
6 & 7 for op1	00	01	10	11
4 & 5 for op2	00	01	10	11
2 & 3 for op3	00	01	10	11

Figure 32. Register flag bits from $INDEX

Error message No. 4 is issued if the number of operand sublist elements is not 1 or 2. Error message No. 5 is issued if an index register other than #1 or #2 is specified.

Registers Used

Software register #2 is used.

BLDTXT Subroutine — Build Object Text

The BLDTXT subroutine builds object text based on a list of object text elements (OTEs). You use the OTE statement to build the object text element.

The BLDTXT subroutine is in the form of copy code. You must include a COPY CBLDTXT statement in your program to use it.

The CALL to the BLDTXT subroutine has the following syntax:

Syntax:

```
label          CALL     BLDTXT
```

Operand	*Description*
BLDTXT	Code BLDTXT as the operand on the CALL instruction.

Entry Conditions

The AMACDATA field in compiler common area must point to a 1-word count of the number of object text elements. You must include the ASMCOMM equates in your program to access the compiler common area. The AMACDATA field must be followed by the object text elements. The length of each OTE is defined by the equate LOTE.

Exit Conditions

None

Registers Used

None

Overlay Program Subroutines *(continued)*

GETVAL Subroutine — Evaluate Character String

The GETVAL subroutine evaluates a character string which is a self-defining term. A self-defining term is a fixed-decimal constant, a hexadecimal constant, or a 1- or 2- byte EBCDIC character string.

Examples of data handled by GETVAL:

- Decimal constants 1, 100, -300, 32767, -12345
- Hexadecimal constants X'12', X'ABCD', X'FFFF', X'1'
- EBCDIC constants C'A', C'XY', C'01', C'E'

The GETVAL subroutine is in the form of copy code. You must include a COPY CGETVAL statement in your program to use it.

The CALL to the GETVAL subroutine has the following syntax:

Syntax:

```
label          CALL      GETVAL,sle,value,errexit
```

Operand	***Description***
GETVAL	Code GETVAL as the first operand on the CALL instruction.
sle	The address of the sublist element (SLE) which points to the string to be evaluated.
value	A word to receive the result of the evaluation.
errexit	The address of an error routine to be branched to if invalid syntax is encountered in the evaluation.

Entry Conditions

None

Exit Conditions

If an error exit is taken, "value" contains the result computed at the time of the error. For example, if the string 123X is evaluated, the result at the time of the error exit is 123.

LABELS Subroutine — Define or Resolve Labels

You use the LABELS subroutine to define or resolve a label for a sublist element (SLE). You can define or resolve the following label types:

- ADDRESS
- EQUATE
- EXTRN
- WXTRN
- ENTRY.

The LABELS subroutine is in the form of copy code. You must include a COPY CLABELS statement in your program to use it.

You code the CALL for the LABELS subroutine differently for label definition and label resolution.

Defining Labels

The CALL for the LABELS subroutine for label definition puts the label you define into the symbol table with the type and value you specify.

The CALL to the LABELS subroutine for label definition has the following syntax:

Syntax:

```
label          CALL      LABELS,#value,#type,1
```

Operand	*Description*
LABELS	Code LABELS as the first operand on the CALL instruction.
#value	The address of the label value to be put into the symbol table.
#type	Label type to be put into the symbol table.
1	Indicates label definition.

Overlay Program Subroutines *(continued)*

Resolving Labels

If you call the LABELS subroutine to resolve a label and the label is defined, the label type and value are returned in #type and #value, respectively. If the label is undefined, an entry is made in the symbol table, and type and value are set to 0 in the symbol table. The #type operand is set to 0, and #value is set to the symbol table pointer index for the symbol.

The CALL for the LABELS subroutine for label resolution has the following syntax:

Syntax:

```
label          CALL       LABELS,#value,#type,0
```

Operand	***Description***
LABELS	Code LABELS as the first operand on the CALL instruction.
#value	The label value is returned here if label is defined; otherwise, the symbol table pointer index for the symbol is returned.
#type	The label type is returned here if label is defined; otherwise, a zero is returned.
0	Indicates label resolution.

Entry Conditions

Software register #1 must point to the SLE of the label to be processed.

Exit Conditions

If a duplicate symbol is encountered in label definition, an error message is issued. You reference the error message number through the #ERRMSG field in the compiler common area. You must include the ASMCOMM equates to refer to this field.

Registers Used

None

MOVEBYTE Subroutine — Move a Byte String

The MOVEBYTE subroutine moves a variable-length byte string to a target location and right pads with blanks.

The MOVEBYTE subroutine is in the form of copy code. You must include a COPY MOVEBYTE statement in your program to use it.

The CALL to the MOVEBYTE subroutine has the following syntax:

Syntax:

```
label          CALL       MOVEBYTE fromsle,toaddr,count
```

Operand	*Description*
MOVEBYTE	Code MOVEBYTE as the first operand on the CALL instruction.
fromsle	The address of the sublist element (SLE) defining the source data.
toaddr	The address of the target location.
count	The size of the target field.

Entry Conditions

None

Exit Conditions

If the number of characters in the SLE is greater than "count", control is passed to ASMERROR. If "fromsle" or the number of characters in the SLE equals zero, the target field is filled with blanks.

Registers Used

None

Overlay Program Subroutines *(continued)*

OPCHECK Subroutine — Check Statement Syntax

You use the OPCHECK subroutine for source statement syntax checking. OPCHECK does the following:

- Compares the number of positional operands in the source statement against the allowable number of positional operands.
- Matches keywords specified in the source against the allowable keywords.
- Stores the operand list element (OLE) and sublist element (SLE) addresses in the $IDEF expansion for each operand coded in the source statement.

The OPCHECK subroutine is in the form of copy code. You must include a COPY COPCHECK statement in your program to use it.

The CALL to the OPCHECK subroutine has the following syntax:

Syntax:

```
label        CALL      OPCHECK,(oplist)
```

Operand	*Description*
OPCHECK	Code OPCHECK as the first operand on the CALL instruction.
oplist	The oplist operand is the label on a $IDEF statement defining the model for an instruction.

Entry Conditions

None

Exit Conditions

Each positional and keyword operand specified in the source statement has its entry in the $IDEF expansion filled in with its OLE and SLE address. If the operand is missing, the corresponding entry in $IDEF is 0.

If an invalid number of positional operands is coded, control passes to the error exit for positional operand errors. This is the label specified (or default) for PERR= on $IDEF. If an invalid keyword is coded, control passes to the error exit for keyword operand errors. This is the label specified (or default) for KERR= on $IDEF.

Registers Used

Software register #2 is used.

Overlay Program Subroutines *(continued)*

SLPARSE Subroutine — Parse Input String

The SLPARSE subroutine divides (parses) an input string into one or more sublist elements (SLEs). The SLEs are separated by commas.

The CALL to the SLPARSE subroutine has the following syntax:

Syntax:

```
label          CALL      SLPARSE,ops,opl,optbl,tblng,n
```

Operand	***Description***
SLPARSE	Code SLPARSE as the first operand on the CALL instruction.
ops	The address of the input string.
opl	The number of characters in the input string.
optbl	The address of the output table to receive the results of the parse routine.
tblng	The length of the table (in bytes) to be generated.
n	The address of an area to receive the number of elements found.

Entry Conditions

None

Exit Conditions

The value of the "n" operand is negative if unbalanced parentheses are encountered in the input string.

Registers Used

None

Chapter 8. Techniques for Improving Performance

This chapter describes some of the techniques you can use to increase performance on the Series/1.

[illegible]

Whenever you reference a data set on a volume, the system searches the data set directory to find the location of that data set on the volume. Assume the volume has several hundred data sets and the data set you need is near the end of the directory. The system has to read each data set directory entry until it finds the data set you need. This searching requires processor time. You can, however, reduce the amount of time it takes the system to search the directory. You do this by arranging the directory to have the frequently used data sets placed at the *beginning* of the directory. You can use the $DIRECT utility (UT or UD commands) to arrange data sets in the directory. Details on how you use the $DIRECT utility are in the *Operator Commands and Utilities Reference.*

Techniques for Improving Performance

How to Get Faster Access to Volumes

Several factors can determine how fast the system can access a volume:

- The order in which you define your DISK statements at system generation
- Whether you define a volume as a "performance" volume
- Whether you define a fixed-head volume on a fixed-head disk.

How and why you might consider using these techniques is described in this section.

How You Should Define DISK Statements

When you define DISK statements at system generation, you should always define (*first*) the device containing volumes you access frequently.

Each device has a volume descriptor entry (VDE) and the VDEs are chained in the order you define the DISK statements. Thus, the system has to read through the VDE chain to locate a volume. If the volume you need resides on the first device disk device you define, the system only has to read the first VDE in the chain.

Specifying Performance Volumes

The system can access a volume designated as a "performance" volume faster than a "nonperformance" volume. You specify performance volumes by coding the VOLNAME= operand on the DISK or TAPE statements at system generation.

Specifying performance volumes saves time because the system records the address of the volume in the volume descriptor entry (VDE) for that device at IPL time. For nonperformance volumes, the system records the volume address in the volume descriptor entry when you load the program.

Each performance volume requires an additional 46 bytes in the supervisor.

Specifying a Fixed-Head Volume

If you have a fixed-head disk, you should always allocate the volume you frequently use in the fixed-head area. Because no "disk seek" operations are required on a fixed-head disk, the system can directly access the volume you need.

You allocate a fixed-head volume by using the $INITDSK utility (AF command). You can allocate one volume in the fixed-head area of the device.

Improving Disk and Tape I/O Performance

You can increase performance for disk and tape I/O operations by coding TASK=YES on each DISK and TAPE statement at system generation. This causes each device to have its own task to service I/O requests as opposed to one task servicing all I/O requests for devices of the same type.

Each DISK or TAPE statement with TASK=YES specified requires an additional 128 bytes in the supervisor.

You can improve I/O performance by using $MEMDISK to allocate all or a portion of unmapped storage to use as a "disk." This disk resembles a single-volume diskette with the volume name of MEMDSK. By placing temporary work data sets on MEMDSK you reduce the amount of time required to access work data sets.

Operator Commands and Utilities Reference describes how to use $MEMDISK in more detail.

How to Speed Up $COMPRES and $COPYUT1

You can reduce the time it takes for $COMPRES or $COPYUT1 operations by requesting dynamic storage. You specify the amount of dynamic storage when you load these utilities. The dynamic storage you specify is the amount of contiguous storage in the partition minus the size of the program(s).

The following is an example of how you request dynamic storage for these utilities:

```
> $L $COMPRES,,2048        (2048 bytes requested)
> $L $COPYUT1,,2048
```

For $COMPRES, maximum performance is reached when you specify dynamic storage as the number of data sets times 32. You can determine the number of data sets by loading $DISKUT1 and issuing the LS command.

For $COPYUT1, the more dynamic storage you request, the greater the performance improvement.

Techniques for Improving Performance

Decreasing $EDXASM Compilation Time

You can reduce the amount of time needed to compile a $EDXASM program by requesting the maximum number of overlays (6) when you load $EDXASM. The default is 4. Specifying the maximum reduces the number of storage loads required by $EDXASM. Use the OVERLAY (OV) option to specify the number of overlays.

You can also reduce the amount of time required to compile or assemble programs by creating temporary work data sets for $EDXASM, $S1ASM, and $EDXLINK. The $MEMDISK utility enables you to create these data sets on the MEMDSK volume. In addition, you can further decrease assembly or compilation time by copying the entire assembler or compiler and all associated overlays onto the MEMDSK volume.

Note: Since MEMDSK is part of the memory system, you will lose the volume in the event of a power failure. Use it only for work data sets, programs, and other files that you can recover if a power failure does occur.

How to Reduce Program Load Time

You can reduce the amount of time it takes the system to load programs by using the $PREFIND utility. You should use $PREFIND in the following instances:

- The program references a large number of data sets or overlays.
- You frequently load the program from disk or diskette.
- The program's environment (data sets/volumes) is not subject to frequent changes.

The $PREFIND utility stores the physical address of all referenced data sets and overlays into the program header. Thus, when you load the program for execution, the system does not have to search volume and data set directories to find the data sets or overlays. For a program requiring a large number of data sets or overlays, the time saving could be significant.

Operator Commands and Utilities Reference describes the use of $PREFIND in detail.

You can use $MEMDISK to reduce the amount of time needed to load programs by placing executable programs and $LOADER on the volume MEMDSK.

Operator Commands and Utilities Reference describes the use of $MEMDISK.

Glossary of Terms and Abbreviations

This glossary defines terms and abbreviations used in the Series/1 Event Driven Executive software publications. All software and hardware terms pertain to EDX. This glossary also serves as a supplement to the *IBM Data Processing Glossary*, GC20-1699.

$SYSLOGA, $SYSLOGB. The name of the alternate system logging device. This device is optional but, if defined, should be a terminal with keyboard capability, not just a printer.

$SYSLOG. The name of the system logging device or operator station; must be defined for every system. It should be a terminal with keyboard capability, not just a printer.

$SYSPRTR. The name of the system printer.

abend. Abnormal end-of-task. Termination of a task prior to its completion because of an error condition that cannot be resolved by recovery facilities while the task is executing.

ACCA. See asynchronous communications control adapter.

address key. Identifies a set of Series/1 segmentation registers and represents an address space. It is one less than the partition number.

address space. The logical storage identified by an address key. An address space is the storage for a partition.

application program manager. The component of the Multiple Terminal Manager that provides the program management facilities required to process user requests. It controls the contents of a program area and the execution of programs within the area.

application program stub. A collection of subroutines that are appended to a program by the linkage editor to provide the link from the application program to the Multiple Terminal Manager facilities.

asynchronous communications control adapter. An ASCII terminal attached via #1610, #2091 with #2092, or #2095 with #2096 adapters.

attention key. The key on the display terminal keyboard that, if pressed, tells the operating system that you are entering a command.

attention list. A series of pairs of 1 to 8 byte EBCDIC strings and addresses pointing to EDL instructions. When the attention key is pressed on the terminal, the operator can enter one of the strings to cause the associated EDL instructions to be executed.

backup. A copy of data to be used in the event the original data is lost or damaged.

base record slots. Space in an indexed file that is reserved for based records to be placed.

base records. Records are placed into an indexed file while in load mode or inserted in process mode with a new high key.

basic exchange format. A standard format for exchanging data on diskettes between systems or devices.

binary synchronous device data block (BSCDDB). A control block that provides the information to control one Series/1 Binary Synchronous Adapter. It determines the line characteristics and provides dedicated storage for that line.

Glossary of Terms and Abbreviations

block. (1) See data block or index block. (2) In the Indexed Method, the unit of space used by the access method to contain indexes and data.

block mode. The transmission mode in which the 3101 Display Station transmits a data data stream, which has been edited and stored, when the SEND key is pressed.

BSCAM. See binary synchronous communications access method.

binary synchronous communications access method. A form of binary synchronous I/O control used by the Series/1 to perform data communications between local or remote stations.

BSCDDB. See binary synchronous device data block.

buffer. An area of storage that is temporarily reserved for use in performing an input/output operation, into which data is read or from which data is written. See input buffer and output buffer.

bypass label processing. Access of a tape without any label processing support.

CCB. See terminal control block.

central buffer. The buffer used by the Indexed Access Method for all transfers of information between main storage and indexed files.

character image. An alphabetic, numeric, or special character defined for an IBM 4978 Display Station. Each character image is defined by a dot matrix that is coded into eight bytes.

character image table. An area containing the 256 character images that can be defined for an IBM 4978 Display Station. Each character image is coded into eight bytes, the entire table of codes requiring 2048 bytes of storage.

character mode. The transmission mode in which the 3101 Display Station immediately sends a character when a keyboard key is pressed.

cluster. In an indexed file, a group of data blocks that is pointed to from the same primary-level index block, and includes the primary-level index block. The data records and blocks contained in a cluster are logically contiguous, but are not necessarily physically contiguous.

COD (change of direction). A character used with ACCA terminal to indicate a reverse in the direction of data movement.

cold start. Starting the spool facility by erasing any spooled jobs remaining in the spool data set from any previous spool session.

command. A character string from a source external to the system that represents a request for action by the system.

common area. A user-defined data area that is mapped into the partitions specified on the SYSTEM definition statement. It can be used to contain control blocks or data that will be accessed by more than one program.

completion code. An indicator that reflects the status of the execution of a program. The completion code is displayed or printed on the program's output device.

constant. A value or address that remains unchanged thoughout program execution.

controller. A device that has the capability of configuring the GPIB bus by designating which devices are active, which devices are listeners, and which device is the talker. In Series/1 GPIB implementation, the Series/1 is always the controller.

conversion. See update.

control station. In BSCAM communications, the station that supervises a multipoint connection, and performs polling and selection of its tributary stations. The status of control station is assigned to a BSC line during system generation.

cross-partition service. A function that accesses data in two partitions.

cross-partition supervisor. A supervisor in which one or more supervisor modules reside outside of partition 1 (address space 0).

data block. In an indexed file, an area that contains control information and data records. These blocks are a multiple of 256 bytes.

data record. In an indexed file, the records containing customer data.

data set. A group of records within a volume pointed to by a directory member entry in the directory for the volume.

data set control block (DSCB). A control block that provides the information required to access a data set, volume or directory using READ and WRITE.

data set shut down. An indexed data set that has been marked (in main storage only) as unusable due to an error.

DCE. See directory control entry.

device data block (DDB). A control block that describes a disk or diskette volume.

direct access. (1) The access method used to READ or WRITE records on a disk or diskette device by specifying their location relative the beginning of the data set or volume. (2) In the Indexed Access Method, locating any record via its key without respect to the previous operation. (3) A condition in terminal I/O where a READTEXT or a PRINTEXT is directed to a buffer which was previously enqueued upon by an IOCB.

directory. (1) A series of contiguous records in a volume that describe the contents in terms of allocated data sets and free space. (2) A series of contiguous records on a device that describe the contents in terms of allocated volumes and free space. (3) For the Indexed Access Method Version 2, a data set that defines the relationship between primary and secondary indexed files (secondary index support).

directory control entry (DCE). The first 32 bytes of the first record of a directory in which a description of the directory is stored.

directory member entry (DME). A 32-byte directory entry describing an allocated data set or volume.

display station. An IBM 4978, 4979, or 3101 display terminal or similar terminal with a keyboard and a video display.

DME. See directory member entry.

DSCB. See data set control block.

dynamic storage. An increment of storage that is appended to a program when it is loaded.

end-of-data indicator. A code that signals that the last record of a data set has been read or written. End-of-data is determined by an end-of-data pointer in the DME or by the physical end of the data set.

ECB. See event control block.

EDL. See Event Driven Language.

emulator. The portion of the Event Driven Executive supervisor that interprets EDL instructions and performs the function specified by each EDL statement.

end-of-tape (EOT). A reflective marker placed near the end of a tape and sensed during output. The marker signals that the tape is nearly full.

enter key. The key on the display terminal keyboard that, if pressed, tells the operating system to read the information you entered.

event control block (ECB). A control block used to record the status (occurred or not occurred) of an event; often used to synchronize the execution of tasks. ECBs are used in conjunction with the WAIT and POST instructions.

Event Driven Language (EDL). The language for input to the Event Driven Executive compiler ($EDXASM), or the Macro and Host assemblers in conjunction with the Event Driven Executive macro libraries. The output is interpreted by the Event Driven Executive emulator.

EXIO (execute input or output). An EDL facility that provides user controlled access to Series/1 input/output devices.

external label. A label attached to the outside of a tape that identifies the tape visually. It usually contains items of identification such as file name and number, creation data, number of volumes, department number, and so on.

external name (EXTRN). The 1- to 8-character symbolic EBCDIC name for an entry point or data field that is not defined within the module that references the name.

FCA. See file control area.

FCB. See file control block.

file. A set of related records treated as a logical unit. Although file is often used interchangeably with data set, it usually refers to an indexed or a sequential data set.

file control area (FCA). A Multiple Terminal Manager data area that describes a file access request.

file control block (FCB). The first block of an indexed file. It contains descriptive information about the data contained in the file.

file control block extension. The second block of an indexed file. It contains the file definition parameters used to define the file.

file manager. A collection of subroutines contained within the program manager of the Multiple Terminal Manager that provides common support for all disk data transfer operations as needed for transaction-oriented application programs. It supports indexed and direct files under the control of a single callable function.

floating point. A positive or negative number that can have a decimal point.

formatted screen image. A collection of display elements or display groups (such as operator prompts and field input names and areas) that are presented together at one time on a display device.

free pool. In an indexed data set, a group of blocks that can be used for either data blocks or index blocks. These differ from other free blocks in that these are not initially assigned to specific logical positions in the file.

free space. In an indexed file, records blocks that do not currently contain data, and are available for use.

free space entry (FSE). An 8-byte directory entry defining an area of free space within a volume or a device.

FSE. See free space entry.

general purpose interface bus. The IEEE Standard 488-1975 that allows various interconnected devices to be attached to the GPIB adapter (RPQ D02118).

Glossary of Terms and Abbreviations

GPIB. See general purpose interface bus.

group. A unit of 100 records in the spool data set allocated to a spool job.

H exchange format. A standard format for exchanging data on diskettes between systems or devices.

host assembler. The assembler licensed program that executes in a 370 (host) system and produces object output for the Series/1. The source input to the host assembler is coded in Event Driven Language or Series/1 assembler language. The host assembler refers to the System/370 Program Preparation Facility (5798-NNQ).

host system. Any system whose resources are used to perform services such as program preparation for a Series/1. It can be connected to a Series/1 by a communications link.

IACB. See indexed access control block.

IAR. See instruction address register.

ICB. See indexed access control block.

IIB. See interrupt information byte.

image store. The area in a 4978 that contains the character image table.

immediate data. A self-defining term used as the operand of an instruction. It consists of numbers, messages or values which are processed directly by the computer and which do not serve as addresses or pointers to other data in storage.

index. In an indexed file, an ordered collection of pairs of keys and pointers, used to sequence and locate records.

index block. In an indexed file, an area that contains control information and index entries. These blocks are a multiple of 256 bytes.

indexed access control block (IACB/ICB). The control block that relates an application program to an indexed file.

indexed access method. An access method for direct or sequential processing of fixed-length records by use of a record's key.

indexed data set. Synonym for indexed file.

indexed file. A file specifically created, formatted and used by the Indexed Access Method. An indexed file is sometimes called an indexed data set.

index entry. In an indexed file, a key-pointer pair, where the pointer is used to locate a lower-level index block or a data block.

index register (#1, #2). Two words defined in EDL and contained in the task control block for each task. They are used to contain data or for address computation.

input buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area for terminal input and output.

input output control block (IOCB). A control block containing information about a terminal such as the symbolic name, size and shape of screen, the size of the forms in a printer, or an optional reference to a user provided buffer.

instruction address register (IAR). The pointer that identifies the machine instruction currently being executed. The Series/1 maintains a hardware IAR to determine the Series/1 assembler instruction being executed. It is located in the level status block (LSB).

integer. A positive or negative number that has no decimal point.

interactive. The mode in which a program conducts a continuous dialogue between the user and the system.

internal label. An area on tape used to record identifying information (similar to the identifying information placed on an external label). Internal labels are checked by the system to ensure that the correct volume is mounted.

interrupt information byte (IIB). In the Multiple Terminal Manager, a word containing the status of a previous input/output request to or from a terminal.

invoke. To load and activate a program, utility, procedure, or subroutine into storage so it can run.

job. A collection of related program execution requests presented in the form of job control statements, identified to the jobstream processor by a JOB statement.

job control statement. A statement in a job that specifies requests for program execution, program parameters, data set definitions, sequence of execution, and, in general, describes the environment required to execute the program.

job stream processor. The job processing facility that reads job control statements and processes the requests made by these statements. The Event Driven Executive job stream processor is $JOBUTIL.

jumper. (1) A wire or pair of wires which are used for the arbitrary connection between two circuits or pins in an attachment card. (2) To connect wire(s) to an attachment card or to connect two circuits.

key. In the Indexed Access Method, one or more consecutive characters used to identify a record and establish its order with respect to other records. See also key field.

key field. A field, located in the same position in each record of an indexed file, whose content is used for the key of a record.

level status block (LSB). A Series/1 hardware data area that contains processor status. This area is eleven words in length.

library. A set of contiguous records within a volume. It contains a directory, data sets and/or available space.

line. A string of characters accepted by the system as a single input from a terminal; for example, all characters entered before the carriage return on the teletypewriter or the ENTER key on the display station is pressed.

link edit. The process of resolving external symbols in one or more object modules. A link edit is performed with $EDXLINK whose output is a loadable program.

listener. A controller or active device on a GPIB bus that is configured to accept information from the bus.

load mode. In the Indexed Access Method, the mode in which records are loaded into base record slots in an indexed file.

load module. A single module having cross references resolved and prepared for loading into storage for execution. The module is the output of the $UPDATE or $UPDATEH utility.

load point. (1) Address in the partition where a program is loaded. (2) A reflective marker placed near the beginning of a tape to indicate where the first record is written.

lock. In the Indexed Access Method, a method of indicating that a record or block is in use and is not available for another request.

logical screen. A screen defined by margin settings, such as the TOPM, BOTM, LEFTM and RIGHTM parameters of the TERMINAL or IOCB statement.

LSB. See level status block.

mapped storage. The processor storage that you defined on the SYSTEM statement during system generation.

member. A term used to identify a named portion of a partitioned data set (PDS). Sometimes member is also used as a synonym for a data set. See data set.

menu. A formatted screen image containing a list of options. The user selects an option to invoke a program.

menu-driven. The mode of processing in which input consists of the responses to prompting from an option menu.

message. In data communications, the data sent from one station to another in a single transmission. Stations communication with a series of exchanged messages.

multifile volume. A unit of recording media, such as tape reel or disk pack, that contains more than one data file.

multiple terminal manager. An Event Driven Executive licensed program that provides support for transaction-oriented applications on a Series/1. It provides the capability to define transactions and manage the programs that support those transactions. It also manages multiple terminals as needed to support these transactions.

multivolume file. A data file that, due to its size, requires more than one unit of recording media (such as tape reel or disk pack) to contain the entire file.

new high key. A key higher than any other key in an indexed file.

nonlabeled tapes. Tapes that do not contain identifying labels (as in standard labeled tapes) and contain only files separated by tapemarks.

null character. A user-defined character used to define the unprotected fields of a formatted screen.

option selection menu. A full screen display used by the Session Manager to point to other menus or system functions, one of which is to be selected by the operator. (See primary option menu and secondary option menu.)

output buffer. (1) See buffer. (2) In the Multiple Terminal Manager, an area used for screen output and to pass data to subsequent transaction programs.

overlay. The technique of reusing a single storage area allocated to a program during execution. The storage area can be reused by loading it with overlay programs that have been specified in the PROGRAM statement of the program or by calling overlay segments that have been specified in the OVERLAY statement of $EDXLINK.

overlay area. A storage area within a program reserved for overlay programs specified in the PROGRAM statement or overlay segments specified in the OVERLAY statement in $EDXLINK.

overlay program. A program in which certain control sections can use the same storage location at different times during execution. An overlay program can execute concurrently as an asynchronous task with other programs and is specified in the EDL PROGRAM statement in the main program.

overlay segment. A self-contained portion of a program that is called and sequentially executes as a synchronous task. The entire program that calls the overlay segment need not be maintained in storage while the overlay segment is executing. An overlay segment is specified in the OVERLAY statement of $EDXLINK or $XPSLINK (for initialization modules).

overlay segment area. A storage area within a program or supervisor reserved for overlay segments. An overlay segment area is specified with the OVLAREA statement of $EDXLINK.

Glossary of Terms and Abbreviations

parameter selection menu. A full screen display used by the Session Manager to indicate the parameters to be passed to a program.

partition. A contiguous fixed-sized area of storage. Each partition is a separate address space.

performance volume. A volume whose name is specified on the DISK definition statement so that its address is found during IPL, increasing system performance when a program accesses the volume.

physical timer. Synonym for timer (hardware).

polling. In data communications, the process by which a multipoint control station asks a tributary if it can receive messages.

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

prefind. To locate the data sets or overlay programs to be used by a program and to store the necessary information so that the time required to load the prefound items is reduced.

primary file. An indexed file containing the data records and primary index.

primary file entry. For the Indexed Access Method Version 2, an entry in the directory describing a primary file.

primary index. The index portion of a primary file. This is used to access data records when the primary key is specified.

primary key. In an indexed file, the key used to uniquely identify a data record.

primary-level index block. In an indexed file, the lowest level index block. It contains the relative block numbers (RBNs) and high keys of several data blocks. See cluster.

primary menu. The program selection screen displayed by the Multiple Terminal Manager.

primary option menu. The first full screen display provided by the Session Manager.

primary station. In a Series/1 to Series/1 attachment, the processor that control communication between the two computers. Contrast with secondary station.

primary task. The first task executed by the supervisor when a program is loaded into storage. It is identified by the PROGRAM statement.

priority. A combination of hardware interrupt level priority and a software ranking within a level. Both primary and secondary tasks will execute asynchronously within the system according to the priority assigned to them.

process mode. In the Indexed Access Method, the mode in which records can be retrieved, updated, inserted or deleted.

processor status word (PSW). A 16-bit register used to (1) record error or exception conditions that may prevent further processing and (2) hold certain flags that aid in error recovery.

program. A disk- or diskette-resident collection of one or more tasks defined by a PROGRAM statement; the unit that is loaded into storage. (See primary task and secondary task.)

program header. The control block found at the beginning of a program that identifies the primary task, data sets, storage requirements and other resources required by a program.

program/storage manager. A component of the Multiple Terminal Manager that controls the execution and flow of application programs within a single program area and contains the support needed to allow multiple operations and sharing of the program area.

protected field. A field in which the operator cannot use the keyboard to enter, modify, or erase data.

PSW. See processor status word.

QCB. See queue control block.

QD. See queue descriptor.

QE. See queue element.

queue control block (QCB). A data area used to serialize access to resources that cannot be shared. See serially reusable resource.

queue descriptor (QD). A control block describing a queue built by the DEFINEQ instruction.

queue element (QE). An entry in the queue defined by the queue descriptor.

quiesce. To bring a device or a system to a halt by rejection of new requests for work.

quiesce protocol. A method of communication in one direction at a time. When sending node wants to receive, it releases the other node from its quiesced state.

record. (1) The smallest unit of direct access storage that can be accessed by an application program on a disk or diskette using READ and WRITE. Records are 256 bytes in length. (2) In the Indexed Access Method, the logical unit that is transferred between $IAM and the user's buffer. The length of the buffer is defined by the user. (3) In BSCAM communications, the portions of data transmitted in a message. Record length (and, therefore, message length) can be variable.

recovery. The use of backup data to re-create data that has been lost or damaged.

reflective marker. A small adhesive marker attached to the reverse (nonrecording) surface of a reel of magnetic tape. Normally, two reflective markers are used on each reel of tape. One indicates the beginning of the recording area on the tape (load point), and the other indicates the proximity to the end of the recording area (EOT) on the reel.

relative block address (RBA). The location of a block of data on a 4967 disk relative to the start of the device.

relative record number. An integer value identifying the position of a record in a data set relative to the beginning of the data set. The first record of a data set is record one, the second is record two, the third is record three.

relocation dictionary (RLD). The part of an object module or load module that is used to identify address and name constants that must be adjusted by the relocating loader.

remote management utility control block (RCB). A control block that provides information for the execution of remote management utility functions.

reorganize. The process of copying the data in an indexed file to another indexed file in a manner that rearranges the data for more optimum processing and free space distribution.

restart. Starting the spool facility w the spool data set contains jobs from a previous session. The jobs in the spool data set can be either deleted or printed when the spool facility is restarted.

return code. An indicator that reflects the results of the execution of an instruction or subroutine. The return code is usually placed in the task code word (at the beginning of the task control block).

roll screen. A display screen which is logically segmented into an optional history area and a work area. Output directed to the screen starts display at the beginning of the work area and continues on down in a line-by-line sequence. When the work area gets full, the operator presses ENTER/SEND and its contents are shifted into the optional history area and the work area itself is erased. Output now starts again at the beginning of the work area.

SBIOCB. See sensor based I/O control block.

second-level index block. In an indexed data set, the second-lowest level index block. It contains the addresses and high keys of several primary-level index blocks.

secondary file. See secondary index.

secondary index. For the Indexed Access Method Version 2, an indexed file used to access data records by their secondary keys. Sometimes called a secondary file.

secondary index entry. For the Indexed Access Method Version 2, this an an entry in the directory describing a secondary index.

secondary key. For the Indexed Access Method Version 2, the key used to uniquely identify a data record.

secondary option menu. In the Session Manager, the second in a series of predefined procedures grouped together in a hierarchical structure of menus. Secondary option menus provide a breakdown of the functions available under the session manager as specified on the primary option menu.

secondary task. Any task other than the primary task. A secondary task must be attached by a primary task or another secondary task.

secondary station. In a Series/1 to Series/1 attachment, the processor that is under the control of the primary station.

sector. The smallest addressable unit of storage on a disk or diskette. A sector on a 4962 or 4963 disk is equivalent to an Event Driven Executive record. On a 4964 or 4966 diskette, two sectors are equivalent to an Event Driven Executive record.

selection. In data communications, the process by which the multipoint control station asks a tributary station if it is ready to send messages.

self-defining term. A decimal, integer, or character that the computer treats as a decimal, integer, or character and not as an address or pointer to data in storage.

sensor based I/O control block (SBIOCB). A control block containing information related to sensor I/O operations.

sequential access. The processing of a data set in order of occurrence of the records in the data set. (1) In the Indexed Access Method, the processing of records in ascending collating sequence order of the keys. (2) When using READ/WRITE, the processing of records in ascending relative record number sequence.

serially reusable resource (SRR). A resource that can only be accessed by one task at a time. Serially reusable resources are usually managed via (1) a QCB and ENQ/DEQ statements or (2) an ECB and WAIT/POST statements.

service request. A device generated signal used to inform the GPIB controller that service is required by the issuing device.

session manager. A series of predefined procedures grouped together as a hierarchical structure of menus from which you select the utility functions, program preparation facilities, and language processors needed to prepare and execute application programs. The menus consist of a primary option menu that displays functional groupings and secondary option menus that display a breakdown of these functional groupings.

shared resource. A resource that can be used by more than one task at the same time.

shut down. See data set shut down.

source module/program. A collection of instructions and statements that constitute the input to a compiler or assembler. Statements may be created or modified using one of the text editing facilities.

spool job. The set of print records generated by a program (including any overlays) while engueued to a printer designated as a spool device.

spool session. An invocation and termination of the spool facility.

spooling. The reading of input data streams and the writing of output data streams on storage devices, concurrently with job execution, in a format convenient for later processing or output operations.

SRQ. See service request.

stand-alone dump. An image of processor storage written to a diskette.

stand-alone dump diskette. A diskette supplied by IBM or created by the $DASDI utility.

standard labels. Fixed length 80-character records on tape containing specific fields of information (a volume label identifying the tape volume, a header label preceding the data records, and a trailer label following the data records).

static screen. A display screen formatted with predetermined protected and unprotected areas. Areas defined as operator prompts or input field names are protected to prevent accidental overlay by input data. Areas defined as input areas are not protected and are usually filled in by an operator. The entire screen is treated as a page of information.

station. In BSCAM communications, a BSC line attached to the Series/1 and functioning in a point-to-point or multipoint connection. Also, any other terminal or processor with which the Series/1 communicates.

subroutine. A sequence of instructions that may be accessed from one or more points in a program.

supervisor. The component of the Event Driven Executive capable of controlling execution of both system and application programs.

system configuration. The process of defining devices and features attached to the Series/1.

SYSGEN. See system generation.

system generation. The processing of defining I/O devices and selecting software options to create a supervisor tailored to the needs of a specific Series/1 hardware configuration and application.

system partition. The partition that contains the root segment of the supervisor (partition number 1, address space 0).

talker. A controller or active device on a GPIB bus that is configured to be the source of information (the sender) on the bus.

tape device data block (TDB). A resident supervisor control block which describes a tape volume.

tapemark. A control character recorded on tape used to separate files.

task. The basic executable unit of work for the supervisor. Each task is assigned its own priority and processor time is allocated according to this priority. Tasks run independently of each other and compete for the system resources. The first task of a program is the primary task. All tasks attached by the primary task are secondary tasks.

task code word. The first two words (32 bits) of a task's TCB; used by the emulator to pass information from system to task regarding the outcome of various operations, such as event completion or arithmetic operations.

task control block (TCB). A control block that contains information for a task. The information consists of pointers, save areas, work areas, and indicators required by the supervisor for controlling execution of a task.

task supervisor. The portion of the Event Driven Executive that manages the dispatching and switching of tasks.

TCB. See task control block.

terminal. A physical device defined to the EDX system using the TERMINAL configuration statement. EDX terminals include directly attached IBM displays, printers and devices that communicate with the Series/1 in an asynchronous manner.

terminal control block (CCB). A control block that defines the device characteristics, provides temporary storage, and contains links to other system control blocks for a particular terminal.

terminal environment block (TEB). A control block that contains information on a terminal's attributes and the program manager operating under the Multiple Terminal Manager. It is used for processing requests between the terminal servers and the program manager.

terminal screen manager. The component of the Multiple Terminal Manager that controls the presentation of screens and communications between terminals and transaction programs.

terminal server. A group of programs that perform all the input/output and interrupt handling functions for terminal devices under control of the Multiple Terminal Manager.

terminal support. The support provided by EDX to manage and control terminals. See terminal.

timer. The timer features available with the Series/1 processors. Specifically, the 7840 Timer Feature card (4955 only) or the native timer (4952, 4954, and 4956). Only one or the other is supported by the Event Driven Executive.

trace range. A specified number of instruction addresses within which the flow of execution can be traced.

transaction oriented applications. Program execution driven by operator actions, such as responses to prompts from the system. Specifically, applications executed under control of the Multiple Terminal Manager.

transaction program. See transaction-oriented applications.

transaction selection menu. A Multiple Terminal Manager display screen (menu) offering the user a choice of functions, such as reading from a data file, displaying data on a terminal, or waiting for a response. Based upon the choice of option, the application program performs the requested processing operation.

tributary station. In BSCAM communications, the stations under the supervision of a control station in a multipoint connection. They respond to the control station's polling and selection.

unmapped storage. The processor storage in your processor that you did not define on the SYSTEM statement during system generation.

unprotected field. A field in which the operator can use the keyboard to enter, modify or erase data. Also called non-protected field.

update. (1) To alter the contents of storage or a data set. (2) To convert object modules, produced as the output of an assembly or compilation, or the output of the linkage editor, into a form that can be loaded into storage for program execution and to update the directory of the volume on which the loadable program is stored.

user exit. (1) Assembly language instructions included as part of an EDL program and invoked via the USER instruction. (2) A point in an IBM-supplied program where a user written routine can be given control.

variable. An area in storage, referred to by a label, that can contain any value during program execution.

vary offline. (1) To change the status of a device from online to offline. When a device is offline, no data set can be accessed on that device. (2) To place a disk or diskette in a state where it is unknown by the system.

vary online. To place a device in a state where it is available for use by the system.

vector. An ordered set or string of numbers.

volume. A disk, diskette, or tape subdivision defined using $INITDSK or $TAPEUT1.

volume descriptor entry (VDE). A resident supervisor control block that describes a volume on a disk or diskette.

volume label. A label that uniquely identifies a single unit of storage media.

Index

The following index contains entries for this book only. See the *Library Guide and Common Index* for a Common Index to all Event Driven Executive books.

Index

Index

IBM Series/1 Event Driven Executive

Publications Order Form

Instructions:

1. Complete the order form, supplying all of the requested information. (Please print or type.)
2. If you are placing the order by phone, dial **1-800-IBM-2468.**
3. If you are mailing your order, fold the order form as indicated, seal with tape, and mail. We pay the postage.

Ship to:

Name: ______

______

Address: ______

______

City: ______

State: ______ Zip: ______

Bill to:

Customer number: ______

Name: ______

______

Address: ______

______

City: ______

State: ______ Zip: ______

Your Purchase Order No.: ______

Phone: (   ) ______

Signature: ______

Date: ______

Order:

Description	Order number	Qty.
Reference books		
Set of the following six books. To order individual copies, use the following order numbers.	SBOF-1627	______
Communications Guide	SC34-0638	______
Extended Address Mode and Performance Analyzer User Guide	SC34-0591	______
Installation and System Generation Guide	SC34-0646	______
Language Reference	SC34-0643	______
Library Guide and Common Index	SC34-0645	______
Messages and Codes	SC34-0636	______
Operator Commands and Utilities Reference	SC34-0644	______
Guides and reference cards.		
Set of the following four books and reference cards. To order individual copies, use the following order numbers.	SBOF-1628	______
Customization Guide	SC34-0635	______
Event Driven Language Programming Guide	SC34-0637	______
Operation Guide	SC34-0642	______
Problem Determination Guide	SC34-0639	______
Language Reference Card	SX34-0165	______
Operator Commands and Utilities Reference Card	SX34-0164	______
Conversion Charts Reference Card	SX34-0163	______
Reference Card Envelope	SX34-0166	______
Set of three reference cards and storage envelope. (One set is included with order number SBOF-1627)	SBOF-1629	______
Binders:		
3-ring easel binder with 1 inch rings	SR30-0324	______
3-ring easel binder with 2 inch rings	SR30-0327	______
Standard 3-ring binder with 1 inch rings	SR30-0329	______
Standard 3-ring binder with 1 1/2 inch rings	SR30-0330	______
Standard 3-ring binder with 2 inch rings	SR30-0331	______
Diskette binder (Holds eight 8-inch diskettes.)	SB30-0479	______

Publications Order Form

Cut or Fold Along Line

Fold and tape Please Do Not Staple Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM Corporation
1 Culver Road
Dayton, New Jersey 08810

Fold and tape Please Do Not Staple Fold and tape

IBM®

International Business Machines Corporation

IBM Series/1 Event Driven Executive
Customization Guide
Order No. SC34-0635-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC34-0635-0
Printed in U.S.A.

Reader's Comment Form

Cut or Fold Along Line

Fold and tape   Please Do Not Staple   Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS   PERMIT NO. 40   ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Information Development, Department 28B
P.O. Box 1328
Boca Raton, Florida 33432

Fold and tape   Please Do Not Staple   Fold and tape

IBM®

International Business Machines Corporation

SC34-0635-0

SC34-0635-0
Program Numbers: 5719-XS5, 5719-ASA
File Number: S1-40
Printed in U.S.A.