DNOS ASSEMBLY LANGUAGE

# Model 990 Computer Programmer's Guide

Part No. 2270508-9701 **
1 August 1981

TEXAS INSTRUMENTS

# MANUAL REVISION HISTORY

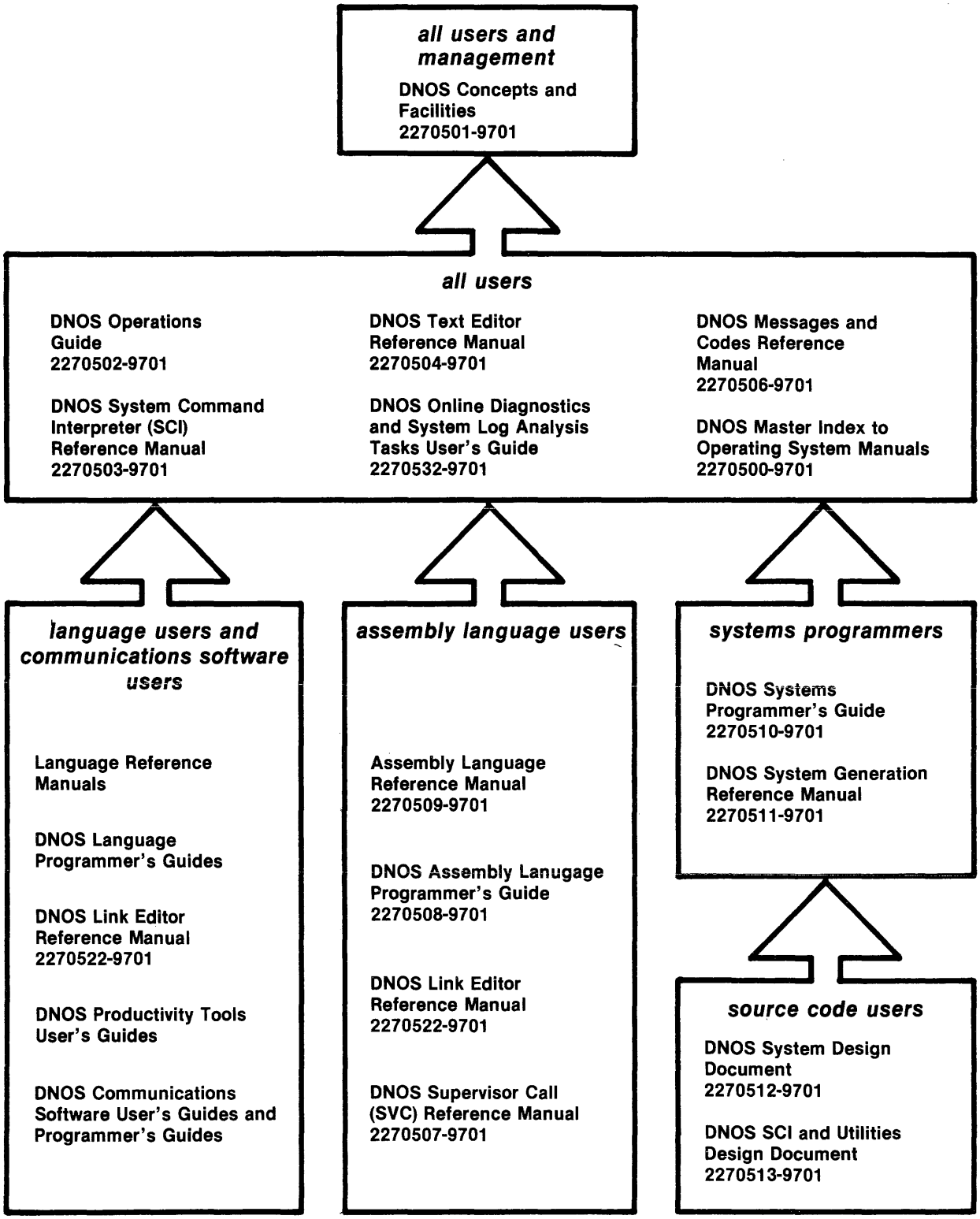
Model 990 Computer DNOS Assembly Language Programmer's Guide (2270508-9701)

Original Issue . . . . . 1 August 1981

The total number of pages in this publication is 206.

# DNOS
## Distributed Network Operating System Software Manuals

The manuals supporting DNOS are arranged in this diagram according to the type of user. The manuals most beneficial to your needs are those contained in the block identified as your user group and in all the blocks above that set.




2280078

# DNOS
# Distributed Network Operating System Software Manuals Summary

**Concepts and Facilities**
Presents an overview of DNOS with topics grouped into functions of the operating system. All new users (or evaluators) of DNOS should read this manual.

**Operations Guide**
Provides the information necessary to perform daily tasks at a TI 990 Computer installation using DNOS. Step-by-step procedures are presented for such tasks as operating peripherals, initializing the system, backing up the system, and manipulating disk files.

**System Command Interpreter (SCI) Reference Manual**
Describes how to use SCI in both interactive and batch jobs. Command procedures and primitives are described, followed by a detailed presentation of all SCI commands in alphabetical order for easy reference.

**Text Editor Reference Manual**
Shows how to use the Text Editor interactively on DNOS and includes a detailed description of each of the editing commands and function keys.

**Messages and Codes Reference Manual**
Lists the error messages, informative messages, and error codes reported by DNOS.

**Online Diagnostics and System Log Analysis Tasks User's Guide**
Provides the information necessary to execute the online diagnostic tasks and the system log analysis tasks and to interpret the results.

**Master Index to Operating System Manuals**
Contains a composite index to topics in the DNOS operating system manuals.

**Programmer's Guides and Reference Manuals for Languages**
Each programmer's guide describes one of the languages supported by DNOS (for example, assembly language, Pascal, COBOL). Each guide covers operating system information relevant to the use of that language in the DNOS environment. The details of the language itself, including language syntax and programming considerations, are in the language reference manual.

**Link Editor Reference Manual**
Describes how to use the Link Editor on DNOS to combine separately generated object modules to form a single linked output.

**User's Guides for Productivity Tools**
Each user's guide describes one of the productivity tools (for example, TIFORM, Query-990, DBMS-990, Sort/Merge) supported by DNOS. Each guide explains the function of the processor, its features, and its interface requirements.

**User's Guides and Programmer's Guides for Communications Software**
Describe the features, functions, and use of the communications software available for execution under DNOS. For example, there is a user's guide for the DNOS 3780/2780 Emulator software package.

**Supervisor Call (SVC) Reference Manual**
Presents detailed information about each DNOS supervisor call and general information about DNOS services.

**Systems Programmer's Guide**
Discusses the DNOS nucleus and subsystems at a conceptual and functional level and describes how to modify the system for a specific application environment.

**System Generation Reference Manual**
Contains the information needed to perform system generation, including pregeneration requirements, generation procedures, and information about postgeneration results.

**System Design Document**
Contains the information needed to understand the functioning of the system when using a source kit. This includes descriptions of the subsystems in detail, naming and coding conventions, module cross-references, data structure details, and information not found in other manuals.

**SCI and Utilities Design Document**
Presents design information about SCI and the DNOS utilities.

# Preface

This manual contains the information necessary for the assembly language applications programmer to assemble, link, and execute programs under DNOS. The manual is designed as a programmer's guide rather than a reference manual so it does not focus on the assembly language instructions. Information in this manual relates to the assembling and executing of programs using the two-pass assembler, SDSMAC.

This manual is organized into the following sections and appendixes:

**Section**

1 Introduction — Presents a brief overview of the steps involved in constructing, assembling, linking, installing, and executing an assembly language program.

2 DNOS Concepts and Environments — Introduces the major concepts and features of the DNOS environment and familiarizes the user with the capabilities of the system.

3 DNOS Assembly Language Program Concepts — Introduces mapping, program segmentation, task attributes, supervisor calls, and file and device services.

4 Building an Assembly Language Program — Provides a brief description of the Text Editor and a sample of the use of the Text Editor commands and editing function keys. Helpful programming techniques are presented for the construction of a source code file.

5 Assembling a Program — Describes the Execute Macro Assembler command and files generated during assembly. Examples of source listing, error message formats, cross-reference listing, and object code are presented.

6 Linking and Installing a Program — Describes the linking operations performed by the Link Editor. Presents samples of linking and installing tasks, procedures, and overlays before execution. Sample link maps and a detailed description of the map contents is given for use in debugging.

7 Executing a Program — Explains the three commands used to execute assembly language programs.

8 Debugging a Program — Presents the debugging commands and samples of debugging techniques.

9 Assembly Language Example — Presents a sample of assembling and executing an assembly language program.

**Appendix**

A Abnormal Completion Messages — Lists the DNOS Abnormal Completion Messages mentioned in Section 5.

B Completion Messages — Lists the DNOS Completion Messages mentioned in Section 5.

C Listing Error Messages — Lists the DNOS Listing Error Messages mentioned in Section 5.

In addition to this manual, the DNOS software manuals shown on the support manual diagram (frontispiece) contain information related to DNOS SVCs. Further manuals containing useful DNOS and assembly language information are listed below:

| Title | Part Number |
|---|---|
| *Model 990 Computer Microcode Development System Programmer's Guide* | 2264445-9701 |

# Contents

## 3 — Assembly Language Concepts

## 4 — Building an Assembly Language Program



## 5 — Assembling a Program

## 8 — Debugging a Program

## 9 — Assembly Language Program Example



## Appendixes



## Index

# Illustrations

# Tables

# 1

# Introduction

## 1.1 DNOS AND ASSEMBLY LANGUAGE

The assembler supported by DNOS is the Model 990 Computer macro assembler (SDSMAC). SDSMAC supports the 990 computer instruction set as well as an extensive macro language capability. In addition to the macro capability, SDSMAC supports the following:

- All instructions of the 990/10 and /12 instruction set with map option
- Thirty-one assembler directives
- Three pseudo-instructions
- Use of parentheses in expressions
- Logical operators in expressions
- Relational operators in expressions
- Many output options
- Workspace pointer directive
- Copy source file directive
- Define operation directive
- Transfer vector pseudo instruction
- Common/Program/Data segment directives

## 1.2 ENTERING PROGRAMS

Assembly language programs may be prepared externally and entered into the system via a card reader or magnetic tape, or they may be prepared at a terminal using the Text Editor to create a file of source code. The compose mode of the Text Editor is used and the source code is entered on a line-by-line basis. Once all of the source code has been entered, the assembly language program is ready to assemble and execute.

Assembly programs may use supervisor calls to perform I/O and program support functions. The supervisor calls are defined in the *DNOS Supervisor Call (SVC) Reference Manual.*

## 1.3 ASSEMBLING PROGRAMS

Assembly language programs are assembled by using the System Command Interpreter (SCI) Execute Macro Assembler (XMA) command procedure. The appropriate entries are made for each request. Once all the entries are made, the assembler is activated. When the assembly has completed, a message appears stating that the assembly is complete. The number of errors or warnings encountered also appears. If errors are detected, the user should consult Appendix A or Appendix B, correct the errors, and reassemble the program.

## 1.4 LINKING AND INSTALLING PROGRAMS

A program must be linked if the assembled program issues references (REFs) to external programs or modules. The Link Editor is defined in detail in the *Link Editor Reference Manual.*

The Link Editor is called by the Execute Link Editor (XLE) command. All modules and libraries to be linked are listed in the link edit control file. The user may also specify the output format.

The output of the Link Editor exists in one of three formats, as defined by the user in the control stream. Two of the formats, normal tagged object and compressed object, are output to a sequential file and must be installed in the system prior to execution. The third format, image, is installed by the Link Editor directly to a user specified program file.

Assembly language programs are installed as procedures, tasks, or overlays by the various installation SCI commands or supervisor calls (SVCs).

## 1.5 EXECUTING PROGRAMS

Assembly language programs can be executed by the Execute Task (XT), the Execute and Halt Task (XHT), or Execute Task and Suspend SCI (XTS) commands, or the various SVCs.

## 1.6 DEBUGGING PROGRAMS

The debugging commands supported by DNOS aid the user in removing errors from (debugging) a program. The debug commands consist of two sets: controlled task commands and commands for all tasks. The controlled task commands operate on tasks in the debug mode. The other set of commands may be used on all tasks. Care must be taken in cases where tasks unconditionally suspend themselves, since some debug commands reactivate tasks.

## 1.7 DNOS RESPONSE NOTATION

Throughout the manual, the System Command Interpreter (SCI) commands are described and discussed for the purpose of aiding the user in the assembly and execution of programs. The legal response type, which may be entered for each particular prompt, is specified in each command description. These response types are listed and defined in Table 1-1.

**Table 1-1. Response Type Indicators**

| Response Type | Definition |
|---|---|
| Pathname | I/O resource pathname. This type includes channel name, devicename, filename and stationname. The pathname may be specified by a synonym, synonym followed by a pathname (synonym.pathname), logical name, or logical name followed by a pathname (logical name. pathname). Legal characters in pathnames include uppercase alphabetic characters, numbers, $, [,], and back slash (\). On 911 VDTs, the back slash character is displayed by pressing the CONTROL and the equal (=) keys. The name must start with an alphabetic character. |
| Devicename | Name of a device (DS01, ST01, etc.). |
| Filename | File name may include disk name, the directory which contains the file, the file name within a directory, a logical name, or a logical name and file name (logical name.file name). |
| Stationname | Station ID (ST01, ST02, etc.). Users can find out the station ID by entering the Show Terminal Information (STI) command. |
| YES/NO | The response to a prompt may be YES, NO, Y, or N. |
| Integer | Hexadecimal or decimal number. Hexadecimal numbers must be preceded by entry of the > symbol or by entry of a leading zero. |
| Integer exp | Decimal or hexadecimal values or expression. Composed of decimal or hexadecimal integers and the operators +, −, *, and /. |
| List | List of decimal or hexadecimal values or expressions, separated by commas. |
| Full exp | Integer expression with the additional operators <, >, and ( ). String operands are also permitted. In debugger controlled mode, symbolic names and the symbols #PC, #WP, #ST, and #Rn are permitted. This type is unique to the SCI debugger. |
| Full exp list | A list of integer expressions separated by commas. |
| Alphanumeric | Alphabetic and/or numeric characters or a dollar sign ($), starting with an alphabetic character. (Used with user IDs, volume names, etc.) |
| Character(s) | Set of any characters. |

To assist the user in determining the range of field prompt responses allowed by DNOS, the notation convention shown in Table 1-2 is used throughout this manual. These notation symbols enclose some prompt responses in the command descriptions to define how DNOS expects the response type to be entered.

**Table 1-2. Field Prompt Notation**

| Notation | Meaning |
|---|---|
| Uppercase | Enter the response as listed. |
| Lowercase | Enter a response of this type. |
| No marks | The response is required. |
| [ ] | The response is optional. |
| { } | The response must be exactly one of the enclosed items or must be a type of one of the enclosed items (choices separated by a slash). |
| Item . . . item | More than one item of this type may be entered to the response. Items should be separated by commas. |
| @ | Synonyms are allowed as responses. |
| ( ) | Represents the initial value. If (*) is shown, the value may be supplied from a synonym set by a previously used command procedure.<br><br>If a list is supplied in a form other than interactively (batch mode or a command procedure calling a command procedure), the list must be enclosed in parentheses. |

# 2

# DNOS Concepts and Environment

## 2.1 INTRODUCTION

This section provides an overview of DNOS and describes some important system capabilities. Although some of these capabilities are not used in program development, they are included to familiarize you with the major system features and concepts. This section includes references to other documentation for more detailed discussion of some topics.

## 2.2 JOB STRUCTURE

DNOS uses a structure of jobs and tasks to perform the functions of a multitasking operating system. This job structure facilitates effective resource usage and subsystem isolation.

A job is a collection of cooperating tasks (programs) initiated by command procedures or from within an executing program. When you log on at a terminal, an interactive job begins. This job is associated with the terminal that started it. When you initiate a batch job, that job is not associated with any particular terminal.

At each terminal, it is possible to have one foreground task and one background task concurrently active in the interactive job. Any number of jobs can be created as batch jobs.

### 2.2.1 Interactive Jobs

An interactive job can include tasks operating in the foreground, in the background, or both. A foreground task can accept data or commands from the terminal as the task operates. However, a background task, although initiated from the terminal, executes without interaction with the terminal until the task is finished. Consequently, you can start a task (for example, updating a data base) in background mode and perform other activities (such as data collection) in foreground mode while the background task is active. When complete, the background task returns a message to the terminal, indicating completion.

Commands entered from interactive terminals are entered in foreground mode. The operating system responds by displaying the appropriate command prompts. Enter the required information; the task now begins execution. While the task executes in foreground, SCI is suspended to avoid interference. User interaction now occurs directly with the foreground task. The *DNOS System Command Interpreter (SCI) Reference Manual* describes the commands used to initiate tasks in all modes.

### 2.2.2 Batch Jobs

Batch jobs use SCI to process batch commands. In the batch mode, SCI accepts commands from any sequentially oriented device (typically a disk file of commands) but not from a terminal. Commands submitted in a batch command stream must include all parameters required for the operation. Also, the commands included must be suitable for execution in the background mode. Commands that initiate operations requiring user interaction (for example, text editing and debugging commands) are not permitted.

## 2.3 USING SCI

The following paragraphs discuss the use of SCI. The *DNOS System Command Interpreter (SCI) Reference Manual* contains complete descriptions of SCI commands, plus procedures for creating new commands and menus.

### 2.3.1 SCI Description

SCI is the interface between you and the operating system, system utilities, the software development programs, and application programs. Application programs can interface with you through user-defined SCI commands and menus.

You can use SCI to activate programs and to pass parameters to the programs during execution. SCI also allows you to build and maintain tables of variables, called *synonyms* and *logical names*, and their values. SCI allows application programs to access these variables for use in the programs.

To execute an application program via SCI, you can use predefined execution commands such as Execute Task (XT), Execute COBOL Task (XCT), and Execute Pascal Task (XPT) or you can write your own SCI command to initiate a program. You can add user-defined commands to the system library, or you can group them in a separate command library. The .USE primitive (described in the *DNOS System Command Interpreter (SCI) Reference Manual)* allows you to specify which command library SCI should use.

You can enter SCI commands from interactive terminals or in batch command streams. In response to commands entered interactively, SCI displays command prompts associated with the command.

When all required prompts have been properly answered, SCI interprets the responses and initiates the requested operation.

### 2.3.2 SCI Activation

The following procedure shows the steps to activate SCI at video display terminals:

1. Turn on the terminal if it is not already on.
2. Press the blank orange key.
3. Press the ! (exclamation mark) key.

4. DNOS responds by displaying or printing the following message:

   DNOS X.X.XX

   where X.X.XX is the release version of DNOS.

5. If user identification is required, DNOS displays the following two prompts:

   USER ID: PASSCODE:

   Type in the assigned user ID and press the RETURN key to signal DNOS that an entry has been made. Next, type in the assigned passcode and press the RETURN key to signal DNOS that an entry has been made. The characters of the passcode entered by the user are not displayed to preserve passcode security.

6. DNOS may respond by displaying the following prompt (if it is not already displayed):

   JOB NAME:

7. Type in a job name and press the RETURN key to signal DNOS that an entry has been made. A job name may be any alphanumeric string (maximum of eight characters) which starts with an alphabetic character or $ and consists of only uppercase characters.

8. DNOS may respond by displaying the following prompt (if it is not already displayed):

   ACCOUNT ID:

9. Type in the assigned account ID and press the RETURN key to signal that an entry has been made.

10. DNOS may respond by displaying the following messages:

    SYNONYM FILE PATHNAME:
    LOGICAL NAME FILE PATHNAME:

11. Type in the pathnames which contain the synonyms and logical names to be used, or press the RETURN key if the default pathnames are to be used.

12. If the job name entered is already in use with the same user ID, DNOS may respond with the following prompt:

    RECONNECT?:

13. Type in YES or NO and press the RETURN key to signal that an entry has been made. YES specifies that this terminal is also to be associated with the job name in use. NO specifies that this terminal is to be associated with a new job.

14. If the log-on is successful, DNOS may respond with the SCI prompt ([ ]) or may display the news file if one exists. SCI then waits for the CMD key to be pressed. After the CMD key is pressed, SCI displays the default main menu and SCI prompt ([ ]) as shown in Figure 2-1. The default main menu may be changed at the option of the systems programmer. Use the .MENU and .OPTION SCI primitives to specify the menu and prompt to be used. Refer to the *DNOS System Command Interpreter (SCI) Reference Manual.*

15. Begin to operate the terminal by entering the SCI commands that are available as determined by the privilege level associated with the user ID. If a command is entered that is not authorized for the user's ID, SCI displays an appropriate error message.

16. While executing SCI commands, the terminal should not be turned off. If the terminal is turned off, device errors are written to the system log and the system may loop in an attempt to complete the command.

```
**************************************************
**       T E X A S   I N S T R U M E N T S      **
**       D N O S   S Y S T E M                  **
**************************************************

Command Groups:

     /DEBUG  - Interactive Debugger
     /DEVICE - I/O Devices
     /DIR    - Directories
     /EDIT   - Text Editor
     /FILE   - File Management
     /JOB    - Job Management
     /LANG   - Language Support
     /LUNO   - Logical Unit Numbers
     /MSG    - Message Facilities
     /NAME   - Synonyms and Logical Names
     /PREXEC - Program Execution
     /PFILE  - Program Files
     /STATUS - Status Reports
     /SYSMGT - System Management
     /VOLUME - Disk Volumes


[]
```

**Figure 2-1. SCI Default Main Menu**

### 2.3.3 Entry of SCI Commands in VDT Mode

To enter an SCI command in VDT mode, type the characters (in uppercase letters) of the command and press the RETURN key. Upon entry of a command, SCI displays the full name of the command entered and all the field prompts associated with the command. Field prompts provide information and request parameters to complete command execution. For example, the following field prompt requests that you identify an output pathname:

```
OUTPUT PATHNAME:
```

### 2.3.4 Examples of Using SCI

The following paragraphs contain examples of specific uses of SCI commands. Consult the *DNOS System Command Interpreter (SCI) Reference Manual* for a complete discussion of the SCI commands.

**2.3.4.1 The Show Background Status (SBS) Command.** Use the SBS command to view the status of a program that is currently executing in background mode and that was initiated from your terminal. Since this command has no associated prompts, the command executes immediately after you enter SBS and press the RETURN key. A message indicating the state of the background activity appears, as follows:

```
[ ]SBS

SHOW BACKGROUND STATUS

I STATUS-1217 TASK IS ACTIVE
```

**2.3.4.2 The List Directory (LD) Command.** Use the List Directory command to list the names of all files and subdirectories in a directory. The display for this command is as follows:

```
[ ]LD

LIST DIRECTORY
                    PATHNAME:  pathname@
          LISTING ACCESS NAME:  [pathname@]
```

In response to the prompt PATHNAME, enter the pathname of the directory whose file names and subdirectory names will be listed. The @ indicates that the pathname can be specified by a synonym.

In response to LISTING ACCESS NAME, enter the pathname of the device or file to which the listing should be written. The brackets ([ ]) indicate that the response is optional. The default value is the terminal at which the command is entered. A null response (pressing RETURN while the cursor is in a blank field) causes the default value to be accepted. In the following case, the directory SYS2.DP0080 is listed to the terminal from which the command was executed.

```
[] LD

LIST DIRECTORY
                PATHNAME: SYS2.DP0080
    LISTING ACCESS NAME:


DIRECTORY LISTING OF:  SYS2.DP0080
  MAX # OF ENTRIES: 101     # OF ENTRIES AVAILABLE: 78

DIRECTORY  ALIAS OF  ENTRIES     LAST UPDATE              CREATION
 ML            *         5      05/30/80  13:44:48  03/17/80 12:51:06
 TIP           *        11      05/07/80  12:02:20  02/11/80 16:44:21


FILE    ALIAS OF  RECORDS     LAST UPDATE      FMT  TYPE    BLK PROTECT
BATCH      *          24    06/03/80  08:16:56  BS   N SEQ   YES
COBOL      *        3550    05/30/80  14:06:46  NBS  N SEQ   YES
DATA       *          17    05/07/80  15:31:57  BS   N SEQ   YES
  .        .           .        .         .      .    .  .    .
  .        .           .        .         .      .    .  .    .
16:21:50 TUESDAY, JUN 03, 1980.
```

### 2.3.5 Batch Use of SCI

To use SCI in a batch mode through batch streams, use the Execute Batch (XB) command; or through a batch job using the Execute Batch Job (XBJ) command. The XB command starts a background task that is associated with your terminal. XBJ starts a new job, not associated with a terminal.

The following paragraphs discuss the characteristics of batch SCI and the differences in format between batch commands and commands entered interactively.

**2.3.5.1 Batch Stream Format.** The first and last commands of a batch stream should be the BATCH and EBATCH commands, respectively. The BATCH command initiates the batch SCI environment. EBATCH indicates that the batch stream contains no more commands to be processed by SCI.

Upon normal completion of the batch stream executing in background mode, the following message appears:

```
BATCH SCI HAS COMPLETED
```

**2.3.5.2 Batch Command Format.** When supplying SCI commands in batch stream format, include the following information for each command:

- The characters of the command
- All required prompts associated with the command
- The parameter values (responses) for the command prompts

The following demonstrates the Execute Link Editor (XLE) command in both interactive and batch form. (Refer to the *Link Editor Reference Manual* for a complete description of the XLE command.)

***Interactive Format.*** When you enter XLE interactively, the following prompts appear:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:  pathname@          (*)
  LINKED OUTPUT ACCESS NAME:  [pathname@]        (*)
        LISTING ACCESS NAME:  [pathname@]        (*)
          PRINT WIDTH (CHARS): [integer]         (80)
```

To execute the command, respond to the CONTROL ACCESS NAME prompt by specifying the pathname of the file or device from which the control stream is to be read. Then, either specify values or accept the default values for the remaining prompts. If the control stream is contained in directory .M, file .CONTROL, the linked output is to be written to directory .M, file .OBJECT, the link editor listing is to be written to directory .M, file .LIST, and an 80-character line is acceptable, respond as follows:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:  .M.CONTROL
  LINKED OUTPUT ACCESS NAME:  .M.OBJECT
        LISTING ACCESS NAME:  .M.LIST
          PRINT WIDTH (CHARS): 80
```

***Batch Format.*** To execute this command in a batch stream, include the characters of the command, all required and any optional prompts that are specified, and the responses to those prompts. The following batch command is equivalent to the interactive version shown previously:

```
XLE    CONTROL = .M.CONTROL, LINKED OUTPUT = .M.OBJECT, LISTING = .M.LIST
```

Notice that the default value for the PRINTWIDTH(CHARS) prompt is accepted by omitting it from the batch command. Also, you can use abbreviated versions of the specified command prompts. The abbreviation must be sufficient to uniquely identify the prompt. Often, only the first character of a command prompt need be entered. For example, the following is equivalent to the previous example:

```
XLE    C = .M.CONTROL, LO = .M.OBJECT, LIST = .M.LIST
```

A batch stream consists of one command or a series of commands in this format when preceded by the BATCH command and followed by the EBATCH command. The file containing the batch command stream is the input file for the XB and XBJ commands. Consult the *DNOS System Command Interpreter (SCI) Reference Manual* for more information on batch command construction and batch capabilities.

**2.3.5.3 Interactive Execution of Batch Streams and Batch Jobs.** Use the XB command to execute batch streams as background activities from an interactive job. After you enter the XB command and the batch stream begins execution, you can continue to execute SCI commands in foreground mode. After the batch stream completes, the completion message appears the next time you press the CMD key. To monitor batch stream execution, you can enter the Show Background Status (SBS) command from time to time or use the WAIT command. Also, you can use the Show File (SF) command to view the listing file for the batch stream during the run.

An example of the XB command is as follows:

```
[ ] XB

EXECUTE BATCH
          INPUT ACCESS NAME:  pathname@
        LISTING ACCESS NAME:  pathname@
```

The INPUT ACCESS NAME is the pathname from which DNOS should read the batch command stream. The LISTING ACCESS NAME is the pathname of the device or file to which DNOS should write the results of the batch stream execution. This device or file must not be used by any command in the batch command stream.

The XBJ command allows you to create the job stream commands and execute a batch SCI job independent of a terminal. Consequently, you can continue to execute SCI commands in foreground or background mode. A description of the XBJ command is as follows:

```
[ ] XBJ

EXECUTE BATCH JOB
                    JOB NAME:  alphanumeric
          USE CURRENT USER ID?:  YES/NO                (YES)
LOGICAL NAME TABLE PATHNAME:  [filename@]
     SYNONYM TABLE PATHNAME:  [filename@]
```

The response to the JOB NAME prompt is a one-to-eight character, user-defined name for the job. If the response to the USE CURRENT USER ID? prompt is NO, a prompt for another user ID appears. (Some installations may require a passcode and/or account ID with the new user ID.) The LOGICAL NAME TABLE PATHNAME is a file containing the logical names to be passed to the new job. The logical name table is created using the Snapshot Name Definition (SND) command, described in the *DNOS System Command Interpreter (SCI) Reference Manual*. To pass the logical names of the creating job, enter a null response (the default). The SYNONYM TABLE PATHNAME is the file name containing the set of synonyms to be used by the new job. (The synonym table is also created using the SND command.) The synonym table must specify the Input Access Name and the Listing Access Name for the XBJ command. As in the XB command, the Input Access Name is the file that contains the batch commands, and the Listing Access Name specifies the file or device to which the results of the batch job should be written. If you enter a null response to the SYNONYM TABLE PATHNAME prompt, DNOS prompts for the INPUT ACCESS NAME and LISTING ACCESS NAME as in the XB command. The *DNOS System Command Interpreter (SCI) Reference Manual* contains further information on the XBJ command.

**2.3.5.4 Entering Programs from Sequential Devices.** You can use any sequential file of program source code for input to the compilers or assembler. If necessary, copy source code that has been key punched on a card deck to a sequential disk file. Program source code, entered by the Text Editor or Copy Concatenate (CC) command, can be read from devices. An example using the CC command to copy the source code from cards to a disk file is as follows:

```
[ ] CC

COPY/CONCATENATE
        INPUT ACCESS NAME(S):  CR01
        OUTPUT ACCESS NAME:    .USER.SOURCE
                   REPLACE?:   NO
    MAXIMUM RECORD LENGTH:
```

## 2.4 DIRECTORY AND FILE STRUCTURE

DNOS file management allows you to build, organize, and access directories and files. A *file* consists of a named collection of data. The data in the file can be generated by you (for example, source code or documentation) or by the system (for example, object code or listing files). A *directory* is a relative record file that contains the information necessary to locate other files and describes the characteristics of those files. It does not contain user data.

### 2.4.1 Establishing Volume Names

*Volume names* are alphanumeric character strings of as many as eight characters that identify the disk on which a file is found. The first character of a volume name must be an alphabetic character. For example, VOL1 could be the volume name of a disk.

The Initialize Disk Surface (IDS) command prepares the disk surface for initialization by the Initialize New Volume (INV) command. The IDS command must be performed prior to the first INV command. It is not necessary to perform another IDS before any further initializations of the disk.

The INV command assigns volume names to disks. Once a volume is initialized by an INV command, all access to files on that volume must include the volume name in the pathname or access name, unless the volume is the system disk or unless a device is specified.

One disk drive on each system (usually DS01) is designated to hold the system disk. The system disk contains all required operating system components, including the loader program, system program files, and temporary system files. The system disk is the default volume when no volume name is specified. For example, .PROOF designates a file named PROOF on the system disk.

### 2.4.2 Establishing Directories

Each disk volume has a file directory named VCATALOG, where DNOS maintains a volume table of contents. The files described in VCATALOG are data files or directory files (Figure 2-2).

DNOS directories contain the names of and pointers to other files. Directories do not contain user data. Typically, related files are contained in a directory. Directories can also contain subdirectories. Both directories and subdirectories are created by the Create Directory File (CFDIR) command. A subdirectory can be created under a directory only after the directory has been created. For example, subdirectory VOL1.SOURCE.PROGRAMA can not be created unless directory VOL1.SOURCE already exists.

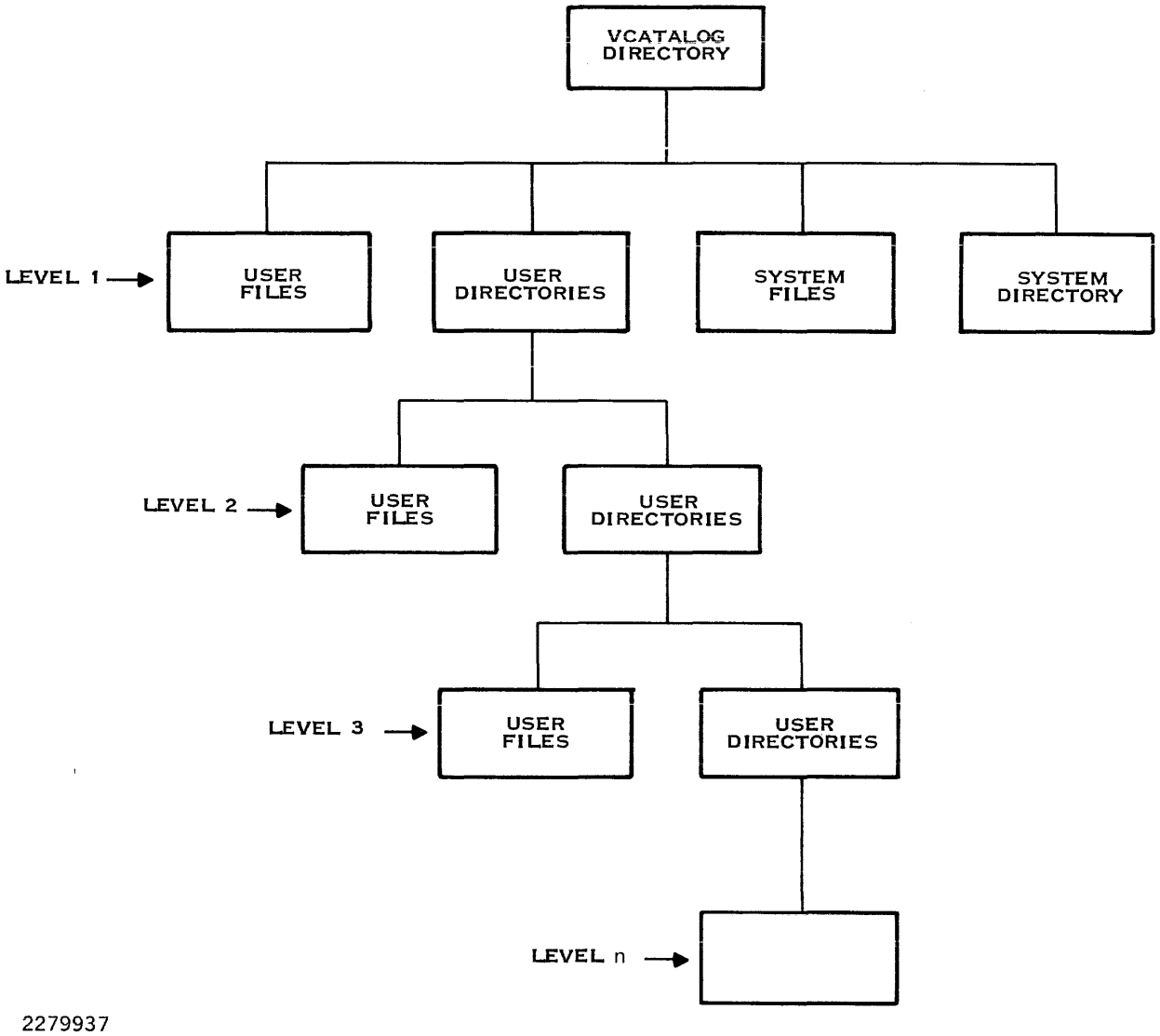



**Figure 2-2. Directory and File Structure**

It is convenient to group related files into a single directory. For example, all source files for a program might be in a directory named VOL1.SOURCE.PROGRAMA; all listings generated from assembly or compilation of source modules for this program might be in a directory named VOL1.LISTING.PROGRAMA. (Refer to Section 3 for more information on alternative ways to structure files for programs.)

Do not assign file names that might be confused with DNOS system file names. Most system file or directory names begin with S$.

### 2.4.3 Establishing Files

After initializing a disk volume and creating directories and subdirectories, you can create files that are accessible either under the volume or under a directory or subdirectory. The following commands are available to create files:

- Create Key Indexed File (CFKEY)
- Create Relative Record File (CFREL)
- Create Sequential File (CFSEQ)
- Create Program File (CFPRO)
- Create Image File (CFIMG)
- Create File (CF)

The CF command requires the subsequent selection of a file type. These commands are described in detail in the *DNOS System Command Interpreter (SCI) Reference Manual.*

## 2.5 PATHNAMES AND ACCESS NAMES

A file on a disk volume is referenced by its pathname. A *pathname* is a concatenation of the volume name, names of the directory levels leading to the file (excluding VCATALOG), and the file name itself. Each component of a pathname cannot exceed eight characters in length. A complete pathname must not exceed 48 characters including periods. The components of the pathname are separated by periods, as in the following examples:

VOL1.AGENCY.RECORDS

MYDIRECT.MYDIRCTA.MYFILE

VOLTWO.DEB

EMPLOY01.USRA.PAYROLL

EMPLOY01.USRB.CATALOGX.PAYROLL

An *access name* may be a device name, volume name or file pathname. For device names, you must use certain default names (except for special devices). Example device names include ST02 for terminal number 2, LP01 for line printer number 1, and DS03 for disk number 3.

You can reference a volume on which a file resides through either the device name or the volume name. Omitting the volume name and beginning the pathname with a period indicates that the file is on the system disk. Samples of valid names for devices and files are as follows:

| File Identifier | Meaning |
|---|---|
| CR01 | Device name |
| DS02.MYCAT.MYFILE | Device name, directory name, file name |
| .MYCAT.MYFILE | System disk, directory name, file name |
| VOLID.MYCAT.MYFILE | Volume name, directory name, file name |

## 2.6 SYNONYMS AND LOGICAL NAMES

DNOS supports use of synonyms and logical names for I/O resources. Synonyms are used to abbreviate long text strings. Logical names are used to abbreviate resource names, define resource access, and pass parameters associated with the resource (devices, files, or channels).

### 2.6.1 Synonyms

*Synonyms* are abbreviations of one or more characters in length that are commonly used in place of long pathnames or portions of pathnames. These synonyms are always available to foreground tasks. Background tasks receive a copy of the foreground synonyms when the background task is initiated. At terminals requiring log-on, user-defined synonyms are associated with that user's ID and are available whenever the user logs on at any terminal. Use the Assign Synonym (AS) and Modify Synonym (MS) commands to define synonyms and to modify defined synonyms. When you enter a synonym in response to an SCI command prompt, the synonym is replaced by the actual text string.

When an SCI command is executed in foreground mode, you can use a synonym only as the first or only component of a pathname (device name or file name). For example, if A is a synonym for directory VOL1.SOURCE and B is a synonym for PROGRAMA in that directory, A.PROGRAMA is an acceptable file name. However, VOL1.SOURCE.B or A.B are not acceptable. Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for use of synonyms in batch streams in the background mode.

### 2.6.2 Logical Names

A *logical name* is a user-specified, alphanumeric string of up to eight characters. Programs use logical names to access I/O resources. An I/O resource can be a device, an IPC channel, a file, or a set of concatenated files. You have the option of assigning a LUNO to a logical name that maps to an access name. (A LUNO is a logical unit number that represents a file or device; see paragraph 2.8.4.)

Since each logical name is associated with a set of parameters (the set assigned to the corresponding I/O resource), logical names provide a means of passing the parameters assigned to a given resource. Use the Assign Logical Name (ALN) command to specify values for these parameters. The *DNOS System Command Interpreter (SCI) Reference Manual* contains a detailed description of this command.

Some examples of the types of parameters associated with logical names are as follows:

- File characteristics
- Access privileges
- Spooler information
- File creation
- Auto-generate pathname
- Job temporary files

## 2.7 FILE TYPES

DNOS supports the following file types: sequential, relative record, and key indexed.

### 2.7.1 Sequential Files

Sequential files are variable-record-length files whose records are always read, written, and accessed serially (that is, record 0 must be accessed first, record 1 must be accessed next, and so on). Some examples of using sequential files are as follows:

- As an input file for card images. If a logical record length of 80 is specified, the sequential file can be treated as a card reader by the program reading the file.
- As an output file. In this function, the file can resemble the line printer.
- As a location for listing files from DNOS processors.

### 2.7.2 Relative Record Files

Relative record files are also called random access files. Unlike sequential files, relative record files may be accessed in any order. Each record has a unique record number, which you specify to access that individual record. The operating system increments the caller's record number after each read or write so that sequential access is permitted. One end-of-file (EOF) record is maintained wherever it was last specified by a program. The range of record numbers is from 0 to one less than the number of records in the file. The maximum number of records in a relative record file is 2 to the 24th power. The records are fixed in length, and the length must be specified during file creation.

Relative record files are useful when each record in the file is already associated with a unique value ranging from 0 to n; for example, in an inventory file, the item number can be specified as the record number. Consequently, information about item number i can be obtained by accessing record number i.

Special types of relative record files available in DNOS are directory, program, and image files. These files provide special interface mechanisms that are used primarily for memory images, memory swapping, and diagnostic dumps.

- Directory Files — Contain names of and pointers to other files
- Program Files — Contain program images and an internal directory of the images
- Image Files — Special-purpose files used primarily by the operating system for memory images, memory swapping, and diagnostic dumps.

### 2.7.3 Key Indexed Files

A key indexed file (KIF) allows random access to its records via a key. The key is a character string of up to 100 characters, located in a fixed position within each file record. From 1 to 14 individual keys may be specified. For example, the records in an employee file might be accessed by keys that indicate the employee's id, name, and social security number.

Keys can overlap one another, with certain restrictions, within the record. Although the keys can be structured anywhere within a record, they must appear in the same relative position in all records in the file. One key must be specified as the primary key; the other keys are secondary keys. The primary key must be present in all records, but secondary keys are optional.

In addition to supporting random access, KIFs include the following characteristics:

- Records can be accessed sequentially in the sort order of any key.
- At file creation, any key can be designated as allowing duplicates, which means that two or more records in the file can have the same value for this key.
- At file creation, any key except the primary key can be designed as being modifiable. This means that when a record is being rewritten, the key value may change. Also, a secondary key value that is missing in the record can be added later on a rewrite.
- Keys can overlap.
- Records can be of variable length and can change in size on a rewrite.
- Searching on partial keys is allowed.
- Records are automatically blank suppressed.
- Record-level locking is supported.
- The size of the file can increase.
- File integrity is maintained through pre-image logging of modified blocks. Before a record is modified on disk, it is copied to a backup area in the file overhead area. Consequently, system failures cause the loss of only the last I/O operation.
- Records of odd or zero length are not allowed.

### 2.7.4 Concatenated and Multifile Sets

Sequential and relative record files can be logically concatenated by setting the values of a logical name to the pathnames of a set of files. Logical concatenation allows access to the files, in sequence, without requiring that they be physically concatenated. (When required, physical concatenation can be performed by the Copy/Concatenate SCI command.) A multifile set is a set of key indexed files, the pathnames of which are the values of a logical name. The files in the set are associated in a nonreversible manner. Individual components of concatenated and multifile sets can be on separate disks.

Several restrictions apply to the concatenation of files. The files must be of the same type and may not be special use files such as directories, program files, key indexed files, or image files. Relative record files to be concatenated must have the same logical record size. A concatenation cannot contain both blocked and unblocked records, and any LUNO assigned to a file must be released before concatenating the file.

The following special rules apply to combining key indexed files in a multifile set:

- At the first definition of the multifile set, all but the first file must be empty.
- None of the files can be a member of an existing multifile set.
- All of the files must have the same physical record size.
- The files must have the same key definitions. In subsequent definitions of these sets, the same files must be associated in the same order, and none of the original set can be omitted. One empty file can be added at the end (but not at any other position).
- You cannot use key indexed file operations to individually access key indexed files of a multifile set. You can access these files only by using operations that examine physical records or absolute disk addresses.

The multifile set of key indexed files permits a larger key indexed file than one disk can store. When a key indexed file can no longer expand because there is insufficient space on the disk, you can create a new file on another disk. By using a logical name, the two files can be used as one. The second file is used as an extension of the first. For example, assume the first file contains 5000 physical records. When physical record 5001 is required, the first physical record of the second file, record 0, is used.

Only a few of the file utility operations of the I/O Operations SVC apply to concatenated and multivolume sets. They are as follows:

| Code | Operation |
|---|---|
| 91 | Assign LUNO |
| 93 | Release LUNO |
| 99 | Verify Pathname |

The Assign Logical Name (ALN) SCI command associates files collectively with a logical name. Actual logical concatenation or creation of a multifile set occurs when a LUNO is assigned to the logical name. You can access a concatenated file only for the duration of the logical name. You must specify the files in the concatenation order desired. You can specify by pathname, synonym,

logical name, or a logical name and pathname combination. However, all forms must resolve to valid pathnames. All files in the concatenation or multifile set must be precreated and online when the logical name is used.

The last file in a concatenation set can be expandable. All other files become nonexpandable until the logical name is released or the job terminates.

When a single end-of-file (EOF) mark appears at the end-of-medium (EOM), the end-of-file is masked. This allows concatenated files to be accessed logically as a single file without the return of intermediate end-of-file marks. Note that any intermediate end-of-file mark not at the end-of-medium is always returned. If two end-of-file marks are encountered at the end-of-medium, a single end-of-file is returned.

Several users can access the same concatenated or multifile set if the access privileges permit. Two concatenated files are identical when they consist of the same pathnames in the same order. To maintain file integrity, an error is returned if any of the precreated files of a concatenated file are being accessed independently. A concatenated file is deleted by deleting the individual files.

## 2.8 I/O FACILITIES

I/O resource management in DNOS allows a program to request resources dynamically during execution. When a resource is requested but is not available, the program or the user is notified immediately. The request for resources is not queued and the program is not suspended. This allows the program to either abort or retry the request, thereby avoiding a deadlock situation.

I/O resources are allocated to programs according to access privileges that the program requests when issuing an open operation. If the requested privilege is compatible with previously granted requests, the open completes without error. The program is then guaranteed the type of access requested (exclusive, exclusive write, shared, or read only).

### 2.8.1 I/O Methods

DNOS supports I/O operations to various types of devices, files, and IPC channels, all of which are referred to as I/O resources. DNOS also supports communication between programs using IPC channels.

Two methods of I/O are available: resource-specific and resource-independent. Resource-specific I/O uses special features of one particular device or file. Resource-independent I/O allows the user to specify I/O for any of several devices without concern for special features. Both types of I/O allow a program to interact with predefined devices, files, and channels. The interaction occurs through the use of LUNOs.

**2.8.1.1 Resource-Specific I/O.** Resource-specific I/O operations assume device, channel, or file peculiarities. For example, activating the graphic capability on the 911 VDT is a resource-specific I/O operation. Other such operations include the following:

- Extended VDT operations
- Create/delete files and other file-specific I/O utility operations
- Direct disk I/O

- Random access operations to key indexed and relative record files
- IPC master-slave channel owner operations

**2.8.1.2 Resource-Independent I/O.** When resource-independent I/O is used, application programs do not distinguish between devices, files, and channels. Also, a program can read and write data records independently of the type of device or file used. Examples of such types of operations include read, write, forward space, and write EOF. All devices, files (including KIF), and channels support resource-independent access.

### 2.8.2 Interprocess Communication

Interprocess communication (IPC) enables two or more tasks to exchange information via communication channels. IPC channels are created by the Create IPC Channel (CIC) command, or the Create IPC Channel I/O SVC. In each channel, one task must be designated as the owner of the channel. The channel owner task controls use of the channel. Requester tasks (slaves) have less flexibility and fewer privileges.

**2.8.2.1 IPC Uses.** IPC is used for four primary reasons:

- Synchronization — Tasks may synchronize activities by passing messages via IPC.
- Queue serving — A channel owner may serve a queue of requests from other tasks.
- Interception — Channel owner tasks receive requests from queues, interpret or modify the information, and pass the changed data to another task or device.
- Messages — Any variety of uses determined by the programs involved.

**2.8.2.2 IPC Channels.** An IPC channel is a logical path used for communications between two tasks. Two types of IPC channels are available in DNOS: master/slave channels and symmetric channels. For a master/slave channel, the owner of the channel (the master) interprets and/or executes messages transmitted on the channel by requesters (slaves). Special commands must be used by the owner to appropriately read and write the messages. For a symmetric channel, the owner and requestor(s) issue simple Read and Write commands. These commands must match each other. The Read command of one task is processed as soon as the other task issues a Write command and vice versa.

**2.8.2.3 Channel Scope.** The scope of a channel governs access to various jobs and tasks. The scope is determined by the channel type: global, job-local, or task-local.

- Global Channel — Not replicated (only one exists in the whole system) and accessible by any task in the system. The channel must first be used by the owner task. The owner task cannot be automatically bid (made ready for execution) by an AL command. Multiple tasks can concurrently use a global channel that permits shared access.
- Job-Local Channel — Replicated once for each job and accessible by any task in the job. The channel can be shared and the owner task may be automatically bid by an AL command.
- Task-Local Channel — Replicated once for each requester task (many per job) in any job. The channel cannot be shared, and the owner must be automatically bid by an AL command from a requester task.

**2.8.2.4 System-Level IPC Functions.** SCI commands are available to perform the following system-level IPC functions:

- Create IPC Channel (CIC)
- Delete IPC Channel (DIC)
- Assign LUNO (AL)
- Release LUNO (RL)
- Show Channel Status (SCS)

**2.8.2.5 Program-Level IPC Functions.** All program-level access to IPC occurs through the use of SVCs. Operations used by a master/slave channel owner are special I/O SVCs; operations used by requesters and by symmetric channel owners are standard I/O SVCs. In general, owner tasks get information from the channels and return an owner-determined response. However, requester tasks use IPC SVCs in a transparent manner; the effect of each call depends on the owner task. Refer to the *DNOS Supervisor Call (SVC) Reference Manual* for more details about channel operations.

### 2.8.3 File I/O

DNOS provides disk file I/O support for application and system programs. Disk file I/O is performed through the same SVC mechanism used to perform I/O to devices. Assembly language programs must directly incorporate the SVC mechanism to perform I/O.

### 2.8.4 Device I/O

A device may be specified by either a device name or by a logical name. All standard DNOS I/O is performed to LUNOs rather than to physical resources. A LUNO, specified in an I/O operation, is a hexidecimal number that represents a file, channel, or device. DNOS maintains a list of LUNOs that indicate corresponding physical devices. LUNOs can be assigned by the AL command, or by use of an Assign LUNO SVC, and can have one of three scopes as follows:

- Global LUNOs are defined (and are available) for all tasks and jobs.
- Job-local LUNOs are defined (and are available) for all tasks in a job.
- Task-local LUNOs are defined only for the task that assigns them.

### 2.8.5 Spooling

The spooling of data can occur during job execution as output is generated by one or more tasks. Spooling is the process of receiving data destined for a particular device (or type of device) and writing that data to a temporary file (or files). The spooler subsystem schedules the printing of job-local and permanent files among available printing devices. You can implement spooling in two ways, either by the PF command, or by sending output to a logical name.

If you use the PF command, specify the following options:

- Banner Sheet — A cover sheet containing the job name, user ID, time, and date.
- Forms — A particular form for printing devices.

- Device Class Type — Any of a class of devices (class name definition). For example, you can specify any line printer, or any printer that prints uppercase/lowercase, without naming a specific printer.

- Format Selection — Either FORTRAN control characters (blank, 0, 1, or + in column one) or ASCII control characters.

- Multiple Copies — Multiple copies for a file or files.

- Priority — Files for printing based on an assigned priority.

To use a logical name , you must assign the logical name, using the ALN command, and specify the options (which are the same as those for the PF command.) You can use the logical name in programs and utility commands, such as SCI, in either batch or interactive mode.

As an example, let's assume you have assigned the logical name OUT and specified the following options:

- LP02

- standard format

- 2 copies

Each time you send a file or listing to OUT, the spooler schedules two copies of OUT to print on LP02 in standard format. You can design strategies according to your specific needs.

## 2.9 SEGMENTS

A task in DNOS consists of various program sections, each of which has certain features (attributes). The attributes of some sections may be different from others. A program section is called a *segment.* A task in DNOS can consist of up to three "segments." The number of segments in a task depends in part on the attributes that can be assigned to the various sections of the program. In general, if all sections of a program have the same attributes, only one segment is needed; if a division of the program is made into sections with differing attributes, multiple segments may be needed.

The user can build the program, specifies appropriate division of the program to the Link Editor, and installs the segments on a program file. The actual movement of segments into memory during execution varies, depending on whether or not the program explicitly requests certain segments. In most cases, DNOS handles segment changes without user action required.

To install a task, specify an initial set of segments (up to three) and the desired mode of access to those segments. To execute a task from an executing program, load the initial segment set (if necessary) and grant the desired access. Use the appropriate SCI command to execute a task from SCI.

## 2.10 MESSAGE FACILITIES

The *DNOS Messages and Codes Reference Manual* describes all system codes and messages in detail and should be consulted if the system displays only the error code. For systems that have the full message displayed, the paragraphs that follow discuss the components of termination messages and two methods of showing expanded error messages. Later sections discuss the use of condition codes and messages in application programs. The *DNOS Systems Programmer's Guide* gives instructions for creating and modifying messages.

### 2.10.1 Error Messages

When an error occurs, SCI displays the message on the bottom line of the terminal screen and inhibits further operation until you acknowledge the message by pressing the CMD key or the RETURN key. Errors may be generated within SCI during SCI command execution or by any utility activated by an SCI command.

The error messages consist of three parts: the error source indicator, a unique identifier, and the message. The error source indicators are as follows:

| Indicator | Meaning |
|---|---|
| I | Informative message |
| W | Warning message |
| U | User error message |
| S | System error message |
| H | Hardware error |
| US | User or system error |
| UH | User or hardware error |
| SH | System or hardware error |
| UHS | User, hardware, or system |

The unique identifier is a code containing the category of the message (such as SVC, Pascal, or utility). This code may be followed by an identifier for a specific message within that category.

For example, if you attempt to access a nonexistent file, the following error message appears:

```
U SVC-0315 filename DOES NOT EXIST (SF; 5)
```

where filename is the name of the file you tried to access. If you need additional information about an error, use on-line expanded error messages or refer to the *DNOS Messages and Codes Reference Manual.*

### 2.10.2 On-Line Expanded Error Message Documentation

If your system supports expanded message information on-line, both the Show Expanded Message (SEM) command and the ? response to the error messages are available.

**2.10.2.1 Show Expanded Message (SEM) Command.** Use the SEM command to display an expanded description of a termination code. Enter SEM to activate the procedure. You are prompted to specify the type of error (such as SVC or SCI) and the message identifier. These appear in the second field of the termination message. An example of the SEM command display is as follows:

```
[ ]SEM

SHOW EXPANDED MESSAGE
          MESSAGE CATEGORY:  SVC
                MESSAGE ID:  0315
```

The following information appears on the terminal:

```
Explanation
The specified file or channel does not exist.

Action
If the file or channel pathname is specified as intended, create the file or channel and retry
the operation. Otherwise, retry the operation specifying the intended pathname.
```

**2.10.2.2 The ? Response.** If you enter a question mark (?) immediately after receiving an error message, SCI uses the error category and message ID to display the expanded description of the error. SCI displays the original message and the same information as the SEM command.

### 2.10.3 Status Messages

Several SCI commands display status messages to inform you of the actions being taken during command execution. These messages appear on the bottom line of the terminal screen. Acknowledge the message by pressing the CMD key or RETURN key so that operation can continue. Expanded status messages can be secured in the same way as error messages.

# 3

# Assembly Language Concepts

## 3.1 INTRODUCTION

A program is a collection of machine instructions and data that direct the activities of the computer to perform a particular function. A program that executes under DNOS is called a task. There may be several activations of the same program at a given time but each activation is a different task. For example, the System Command Interpreter (SCI) is a program and each station may have, as a task, a unique activation of the SCI program. A program becomes a task when DNOS assigns a runtime ID. Multiple copies of a task may share common procedure or data segments.

## 3.2 PROGRAM MAPPING

The 990/10 and 990/12 computer hardware has a 20-bit memory address bus and can address 1024K words of memory. The logical address space available to a task is limited, by a 16-bit byte address, to 64K bytes. This difference is resolved by DNOS mapping the logical address space of the task into the physical address space of the computer. As shown in Figure 3-1, the mapping hardware can map one, two, or three segments of logical address space into one, two, or three segments of physical address space.

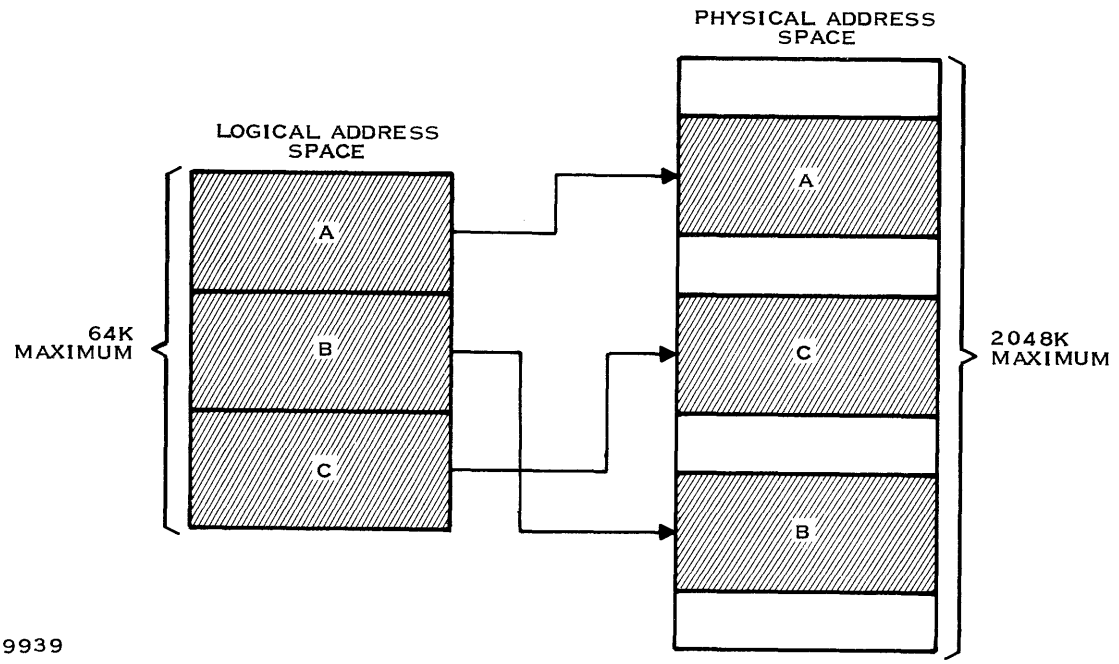



**Figure 3-1. Mapping**

The segments in physical address space need not be contiguous. Since DNOS maintains separate mapping parameters for each task, each task may consist of one, two, or three segments with a total size of 64K bytes. A program includes one unique segment called a task segment. The task segment must contain the workspace address, the entry point, and the end action entry point. A program may also contain sharable segments called procedures. Figure 3-2 illustrates two tasks sharing two segments of memory. The two tasks could be, but need not be, instances of the same program. For example, both tasks might be instances of the SCI program executing at different stations.

The 990 computer instructions which control mapping are privileged. (Use of these instructions by nonprivileged user tasks causes task termination.) DNOS memory management controls mapping so that the mapping function is transparent to user tasks.

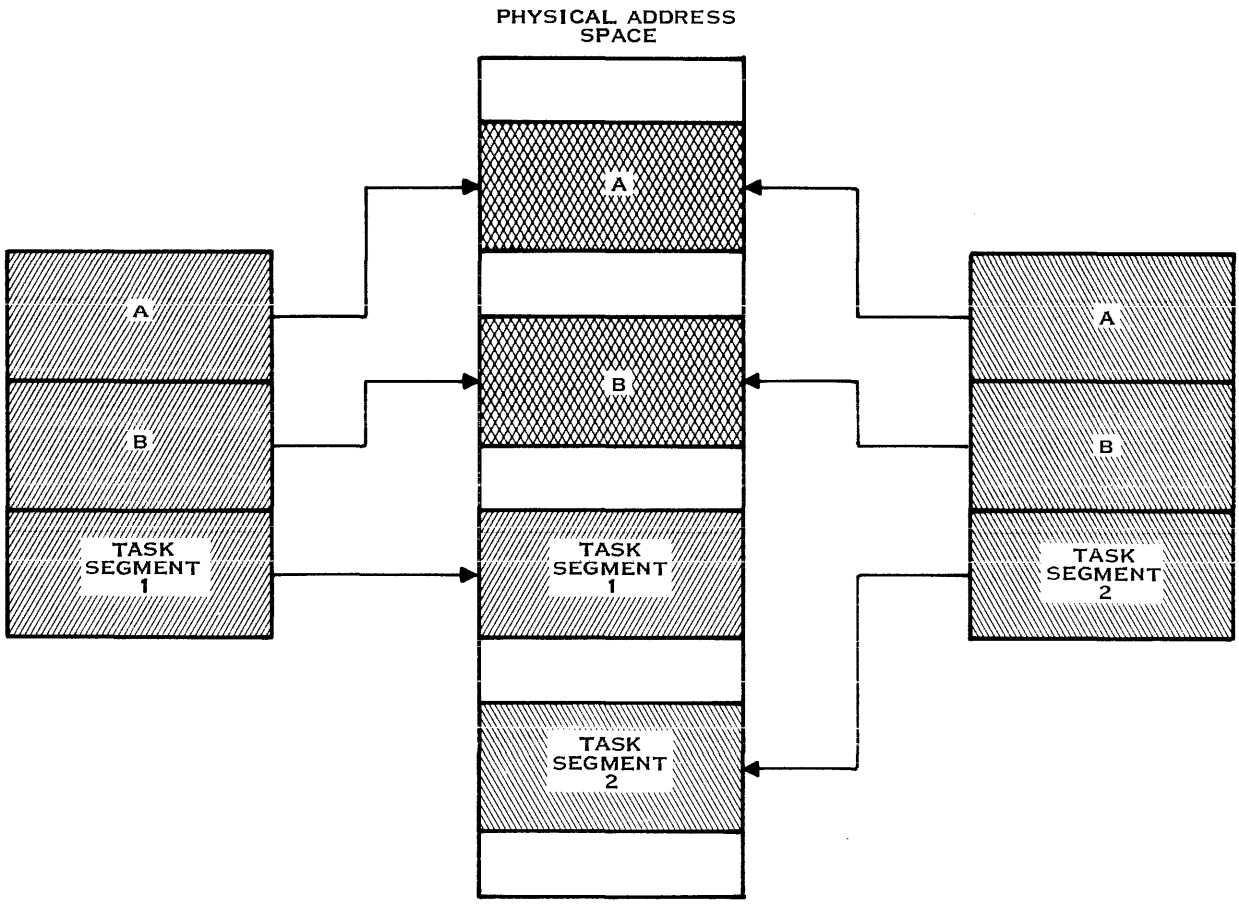



**Figure 3-2. Tasks Sharing Segments**

## 3.3 PROGRAM SEGMENTATION AND PROCEDURAL STEPS

Mapping allows users to segment programs as:

- Single segments, including both data and executable code. When installed on a program file, these segments are called task segments, and each instance of the program in execution is called a task.
- Two separately loadable segments consisting of a procedure segment and a task segment.
- Three separate segments, consisting of two procedure segments and a task segment.

Since DNOS manages memory in 32-byte blocks, the following boundary rules apply for programs consisting of two or three separate segments:

- The first procedure segment (if any) begins at address 0 in the logical address space seen by the executing program.
- The second procedure segment (if any) begins on the first 32-byte boundary immediately following the first procedure in the logical address space seen by the executing program.
- The task segment begins on the 32-byte boundary immediately following the last procedure in the logical address space seen by the executing program.

These memory boundary requirements are supported by the Link Editor, as defined in the *Link Editor Reference Manual.*

Figure 3-3 illustrates the possible memory configurations for programs under DNOS. The following paragraphs outline the processes for achieving any of these configurations. The maximum allowable memory for each task in any configuration is 64K bytes.

### 3.3.1 Single Task Segment

Part A of Figure 3-3 illustrates a program consisting of a task segment. The following process generates this configuration:

1. The source program is assembled (perhaps in several modules).
2. If more than one object module is involved, the object code is linked into one module by the Link Editor.
3. The linked object module is installed on a program file as a task segment using the Install Task (IT) command, or as a function of the Link Editor FORMAT IMAGE command.
4. The installed task is executed using an SCI command or SVC operation.

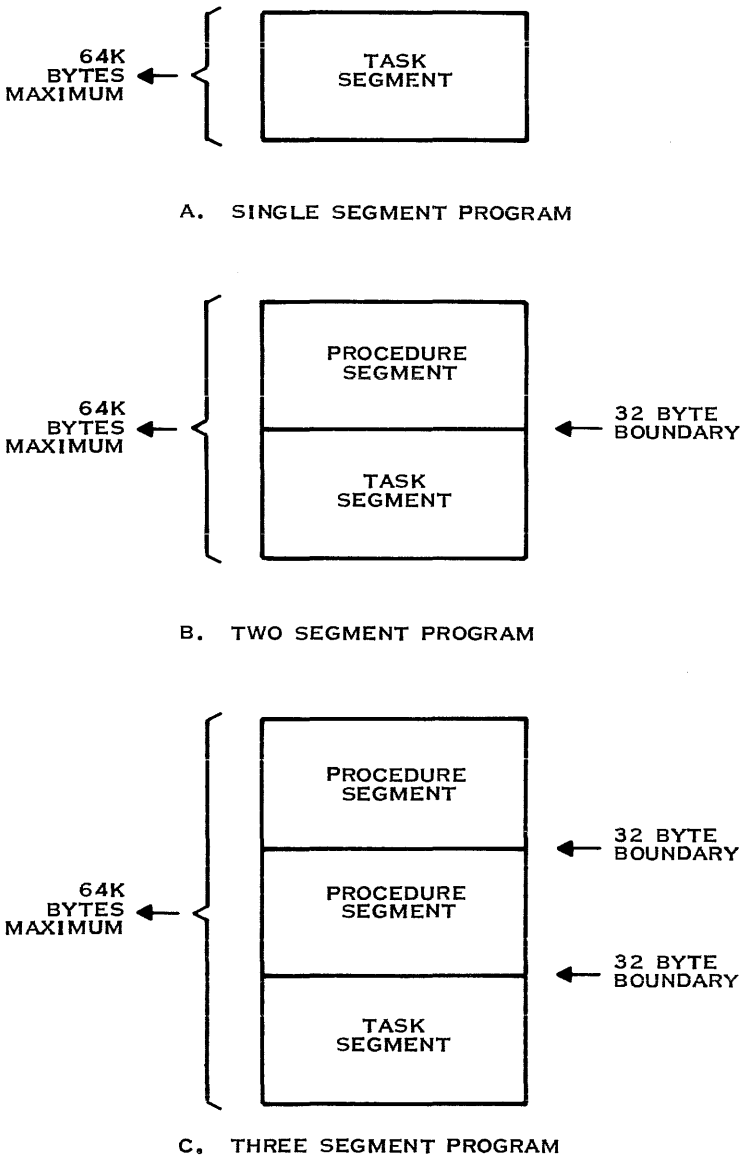



**Figure 3-3. Task Memory Configurations**

### 3.3.2 Task Segment and One or Two Procedures

Parts B and C of Figure 3-3 illustrate a program with a task segment and procedure segment(s). The following process generates this configuration.

1. The various modules of the program are separately assembled.

2. The two (or three) segments of the program are linked by the Link Editor. To be separately loadable, the procedures must be specified by the PROCEDURE command to the Link Editor. The task must be specified with a TASK or PHASE 0 command.

3. The procedure(s) is installed using the Install Procedure (IP) command or by the Link Editor FORMAT IMAGE command.

4. The task segment is installed using the IT command with the ATTACHED PROCEDURES? option selected. The procedure IDs specified during the IP command are entered in response to the prompts of the ATTACHED PROCEDURES? option prompts.

5. The installed task is executed using an SCI command or SVC operation.

## 3.4 SUPERVISOR CALLS

The interface between application programs and DNOS is the supervisor call (SVC). In programs written in a high level language, most supervisor calls are provided in the run-time package for the language and are transparent to the user. However, the high level language programmer may write SVCs to perform functions not otherwise available. The assembly language programmer must code all supervisor calls.

Supervisor calls are implemented as an extended operation (XOP) in DNOS. Specifically, XOP 15 is the means of entry to the SVC processor of DNOS. The address supplied with the XOP instruction is that of the user's supervisor call block. The following is an example of the code for a SVC:

```
XOP     @SCB,15     EXECUTE SVC DEFINED IN BLOCK SCB
```

The assembly language includes a directive that provides a convenient and meaningful substitute for the XOP command. The DXOP directive defines a symbolic operation code for an XOP. The following example defines SVC as XOP 15:

```
DXOP    SVC,15      DEFINES SVC AS XOP 15
```

When you include the DXOP at the beginning of your program, you may code an SVC as follows:

```
SVC     @SCB        EXECUTE SVC DEFINED IN BLOCK SCB
```

## 3.5 THE SUPERVISOR CALL BLOCK

The DNOS supervisor call block is the data structure that defines the supervisor call. The statements described in the preceding paragraph apply to all DNOS supervisor calls. The difference between SVCs is the content and format of the supervisor call block.

A supervisor call block consists of at least one byte, and as many additional bytes as the SVC requires. The first byte (and only byte for some SVCs) contains the opcode that defines the SVC. Opcodes >00 through >7F are reserved for SVCs supported by DNOS. The user may define SVCs for applications in the range of >80 through >FF. Creation of user-defined SVCs is described in the *DNOS Systems Programmer's Guide.*

The second byte of many SVCs is the return code byte. DNOS returns a satisfactory completion code (zero in most cases) in this byte when the operation completes successfully. DNOS returns an error code in this byte when the operation completes in error.

Some of the DNOS SVCs provide several different operations as determined by a sub-opcode in the third byte of the supervisor call block. In these cases, the actual operation to be performed is selected by the opcode and the sub-opcode.

Some of the bytes in the supervisor call blocks of some SVCs contain the result of the requested operation after the operation has completed. That is, the system returns values in some fields of some supervisor call blocks.

The additional bytes of the call blocks of SVCs may contain various types of information related to the operation which are provided by the user:

- Flags
- Input or output data
- Addresses of input or output data
- Size or count values
- Identifiers
- Task parameters
- LUNOs
- Character strings

The specific supervisor call block for each SVC is described in the *DNOS Supervisor Call (SVC) Reference Manual*.

## 3.6 ENTRY VECTOR

DNOS transfers control to a task via an entry vector in the task segment. The first three words of the task segment must consist of the following vector:

| | |
|---|---|
| First Word: | Initial workspace address (WP). |
| Second Word: | Initial entry point address (PC). |
| Third Word: | End action entry point address. |

For example:

```
IDT 'TASK'
REF PROC1, ERPROC
DATA WSPACE
DATA PROC1
DATA ERPROC
WSPACE  BSS 32 WORKSPACE
          .
*         Task Data
*           .
*           .
          END
```

If the end action entry point address is zero, DNOS terminates the task when a fatal error is detected. If the address is not zero, control is transferred to the specified address whenever a fatal task error occurs. If no end action routine exists or when the end action routine executes an End Task SVC, DNOS releases resources from the task and takes it out of execution.

## 3.7 SHARING PROCEDURE SEGMENTS

A program may have one or two procedure segment(s) which contain code or data to be shared with a second activation of the program or with a different program. In the case where two or more tasks using the procedure segment are active concurrently, the procedure segment must be reentrant to be sharable (refer to the paragraph on reentrant programming for more information). If there will never be two or more tasks active at the same time using the procedure segment, the procedure segment may contain modifiable code or data storage and still be sharable.

To be reentrant, a procedure segment must not contain any data storage that is modified during execution, including a workspace. Also, all tasks that include a shared procedure segment must be linked so that any memory addresses in the task segment that are referenced by the procedure segment contain the same code or data storage in each individual task segment.

Refer to section on linking and the *Link Editor Reference Manual* for command descriptions and further linking information.

## 3.8 REENTRANT PROGRAMMING

A procedure is called reentrant if it may be shared by several users at the same time without loss of data. A procedure is called pure if it contains only unmodifiable code and constant data; it is, effectively, read only. Pure procedures are reentrant. Moreover, every program may be split into a pure part containing only executable code and another part containing data and code which is modified at run time.

DNOS supports shared pure procedures. (Nonreentrant procedures may be shared, but DNOS does not guarantee data integrity.) Further, the 990 language processors and the 990 Link Editor support a special construction which simplifies reentrant coding. That is, if an assembly language program uses the Program Segment (PSEG) directive to denote executable code, the Common Segment (CSEG) directive to denote common data, and the Data Segment (DSEG) directive to denote locally used data for modules included in a procedure, the Link Editor separates the program accordingly:

- Executable code into the procedure segment
- Data into the task segment
- Common data into the task segment

If the PROCEDURE directive is used during link edit, the procedure segment created during linking contains no DSEGs. If all the volatile storage is in DSEGs, then the procedure will be reentrant.

A shared procedure must either be installed on the program file on which the task is installed, or on the shared program file, .S$SHARED.

For further information, consult the *Link Editor Reference Manual.*

## 3.9 OVERLAYS

Overlays are part of a program that resides on disk until explicitly requested. When requested, the overlay replaces part of the program previously in memory. The use of overlays can reduce the amount of memory required by a program to the amount required for the largest segment needed at one time. Programs that do not use overlays are loaded in their entirety into memory for execution.

In the subsequent discussion of overlays, the following definitions apply:

- Phase — The smallest functional unit that can be loaded as a logical entity during execution. The linked object output by the Link Editor contains one object module for each phase.
- Root Phase — The main or memory resident phase (level 0) of a program.
- Level — The point at which a new phase begins, identified by a level number in the overlay structure. Phases having the same level number and same parent (preceding) phase are loaded at the same location and are mutually exclusive (they cannot be in memory at the same time).
- Path — A series of phases starting with the root phase and including a successive, higher numbered phase at each level.

The memory requirements of a program can be reduced significantly if it is split into two or more overlays that share the same overlay area. A root phase that is not overlaid must be added to the program to control the loading of the overlays. However, overlays increase execution time by the time required to load the overlays. An effective scheme is to overlay seldom-used phases of a program and retain common code in the root phase.

### 3.9.1 Overlay Structures

Overlay structures are defined by the user and generated by the Link Editor. The structure of an overlaid program is dependent upon the relationships between the phases in the program. Phases that do not have to be in memory at the same time can overlay each other. These phases are considered to be independent in that they do not reference each other either directly or indirectly. Independent phases can be assigned the same load address and are loaded only when referenced. Refer to the *Link Editor Reference Manual* on organizing overlay structures.

### 3.9.2 Overlay Loading

In the root segment or in phases in memory, the user loads overlays from a program file into memory with the Automatic Overlay Manager or by issuing the Load Overlay SVC call.

When the Link Editor is used to produce overlaid programs, the user can specify (by use of the LOAD command) that the Link Editor include an Automatic Overlay Manager in the linked output. The overlay manager performs automatic overlay loading during execution of the overlaid program. The LOAD command is only applicable when the IMAGE format is selected.

### 3.9.3 Relocatable Overlays

Overlays are usually loaded into memory at the load address determined during link edit. However, a user may load an overlay elsewhere in memory. Modules installed as relocatable overlays in a program file can be relocated by the Load Overlay SVC. This enables users to load overlays where space is available rather than where link edited. However, the relocatable overlay cannot have references to other overlays.

## 3.10 SEGMENT MANAGEMENT

In addition to program segments already described, disk- or memory-based program segments may be provided. These program segments may be dynamically mapped into or removed from the memory area of the task.

Disk-based program segments are installed on program files using the Install Procedure/Program Segment SVC. They are loaded into memory and mapped into memory areas of tasks by the segment manager. The segment manager maintains a count of the tasks that require the program segment, and disposes of program segments that are no longer required. When the program segment contents have not changed, or when changes to the disk copy are not permitted, segment manager releases the memory space occupied by the program segment. Otherwise, segment manager writes the program segment to the program file and releases the memory space.

A task may request the segment manager to hold a program segment in memory even when no task currently requires the program segment. This allows passing data to another task that may not be currently executing. When the program segment is no longer required, a task may request the segment manager to release the program segment.

Memory-based program segments are created by the segment manager as uninitialized program segments, and are mapped into the memory area of the task that requests their creation. The task then writes the required data into the program segment. Shared memory-based program segments are a means of passing data between tasks in the same job or in different jobs. A task must reserve this type of segment or the memory is released when the segment is released.

Attributes are defined for a segment when it is installed. The attributes of a task segment are specified in the Install Task SVC that installs the task on a program file. Similarly, the attributes of a procedure segment or a disk-based program segment are specified in the Install Procedure/Program Segment SVC. The attributes of a memory-based program segment are specified when the program segment is created by the segment manager. The segment attributes are as follows:

- Readable. The segment may be accessed in memory for read operations.
- System. The segment may only be accessed by system tasks.
- Memory resident. The segment remains accessible in memory.
- Replicatable. More than one copy may exist in memory.
- Share protected. Segment may not be shared concurrently by two or more tasks.

- Writable control store. Segment contains executable code that accesses writable control store.
- Execute protected. Segment contents may not be executed.
- Write protected. Segment contents may not be altered in memory.
- Updatable. Segment will be written to its permanent file position on disk if it has been marked modified.
- Reusable. Segment may be used consecutively without reloading. This segment may reside on the software cache list while memory space is available for it.
- Copyable. Segment may be replicated by copying the segment from the memory copy.
- Privileged. Segment has been installed in a program file as a privileged task segment. The task can execute hardware privileged instructions.
- Software privileged. Segment has been installed in a program file as a software privileged task segment. The task can issue privileged supervisor calls.
- Overflow protected. Segment has been installed in a program file with overflow protection. During execution, arithmetic overflow is detected as a fatal task error.

## 3.11 TASK ATTRIBUTES

Task attributes are determined during task installation in response to prompts during the IT command or the Install Task SVC. The attributes are:

- Privileged (Hardware and Software)
- System
- Priority
- Memory-Resident
- Replicatable
- Protected, Delete and Execute (990/12 only)
- Copyable
- Reusable
- Updatable
- Arithmetic Overflow Protected (990/12 only)
- Writable Control Storage (990/12 only)

### 3.11.1 Privileged

There are two kinds of privilege: hardware and software. The Link Editor cannot be used to install privileged tasks on a program file. These tasks must be installed using the Install Task SVC, or the IT command.

**3.11.1.1 Hardware Privileged.** The PRIVILEGED? prompt of the IT command or the corresponding flag for the Install Task SVC indicates whether or not a task may issue privileged instructions. Only tasks installed as privileged tasks may issue privileged hardware instructions. Most programs are nonprivileged and may not use the following 990 Computer assembly language instructions:

- Computer Control Instructions (RSET, IDLE, LREX, LIMI)
- Real Time Clock Instructions (CKON, CKOF)
- Mapping Instructions (LMF, LDD, LDS)
- I/O Instructions (SBO, SBZ, TB, LDCR, STCR) when the communications register unit (CRU) address is greater than >E00.

**3.11.1.2 Software Privileged.** Task specified as having software privileges may execute all system SVCs. The SOFTWARE PRIVILEGED?: prompt of the IT command or the corresponding flag for the Install Task SVC commands determine privilege. Nonprivileged tasks may not use the following SVCs:

Install Disk Volume
Unload Disk Volume
Initialize Disk Volume
Disk Management
Direct Disk I/O
Open File Unblocked
Install Task
Install Procedure/Segment
Install Overlay
Delete Task
Delete Procedure/Segment
Delete Overlay
Kill Task
Read/Write TSB
Read/Write Task
Assign Space on Program File
Initialize Time and Date

### 3.11.2 System

A task is either a system task or a user task. System tasks execute in system memory address space (coexistent with other portions of the system).

A task is defined as a system task by the Install Task SVC or by the user responding YES to the SYSTEM TASK? prompt in the IT command during task installation. The Link Editor cannot be used to install a system task.

### 3.11.3 Priority

DNOS requires that each task be installed on a program file at a defined priority level. DNOS provides the following 132 levels of installed priority:

| | | |
|---|---|---|
| (Highest) | 0 | Critical System Tasks (Reserved for DNOS) |
| | R1-R127 | Real-time priorities |
| | 1 | Foreground interactive tasks |
| | 2 | Foreground compute-bound tasks |
| (Lowest) | 3 | Background tasks |
| (Floating) | 4 | (Priority 1 and 2) |

Priority level 0 is intended for the most critical system functions and is reserved for DNOS internal use only. The remaining system tasks are distributed appropriately among the lower priority levels with regard to their relative importance.

Real-time priorities provide the user the capability to supercede all except the most important system tasks. For applications which require prompt access to the CPU, DNOS forgoes some routine maintenance of system duties in an effort to schedule real-time tasks.

Priorities 1, 2, 3, and 4 satisfy requirements of most installations. Priority levels 1 and 2 are used mainly by programs requiring user interaction. Priority level 1 gives quick response for programs interacting with the user's terminal, while priority level 2 is adequate for programs requiring multiple disk accesses.

For programs requiring user interaction and multiple disk accesses, execution priority level 4 automatically switches between priority levels 1 and 2 as the program executes. Therefore, programs requiring user attention can be serviced quickly.

Priority level 3 is for programs executing in batch or background mode, which require no user interaction.

The priority level of a task is assigned during task installation in response to the PRIORITY? prompt of the IT command or the Install Task SVC. The real-time task priorities are assigned with the Install Real-Time Task (IRT) command or the Install Real-Time Task SVC.

DNOS executes tasks according to their priority level by allocating the CPU to the highest priority task awaiting execution. As newly-bid tasks enter the system, the tasks awaiting execution are requeued according to the newly-bid task priorities.

If time slicing is enabled, tasks execute for a fixed interval of time. Upon expiration of a time slice, the task is rescheduled. Time slicing allows tasks of equal priority to share the CPU in a round-robin fashion. (Time slicing and slice value are determined during system generation).

### 3.11.4 Memory-Resident

Tasks may be memory resident (always in memory) or disk resident (in memory when active, but possibly swapped to the disk when not active). Most user tasks should be disk resident to free memory for other tasks, since some supervisor calls depend on the DNOS swapping facility. Memory-resident tasks must be installed on the utilities program file .S$UTIL, or the shared program file .S$SHARED, or on the applications program file defined during system generation.

These tasks are disk-resident tasks until an Initial Program Load (IPL) is performed. Certain support features which depend on swapping (such as dynamic memory allocation) are not available to a memory-resident task.

A memory-resident task, even if terminated, occupies memory until deleted from the program file and an IPL is performed.

Memory-resident tasks are installed on program files by the IT command or the Install Task SVC. The Link Editor cannot be used to install memory-resident tasks on a program file. Tasks are deleted from program files by the Delete Task (DT) command or the Delete Task SVC. Task residency is determined during installation in response to the MEMORY RESIDENT? prompt.

### 3.11.5 Replicatable

Tasks specified as replicatable may have multiple copies concurrently in memory. Replicatable tasks are frequently used in multiterminal environments or in industrial applications where several similar device types are controlled. A replicatable task allocates multiple copies of a task in memory simultaneously and conserves disk space and time. Allocating tasks in this way, avoids the requirement of installing a copy of the same task with a different ID for each concurrent activation of the program. Replicatability is determined by the installation response to the REPLICATABLE? prompt.

### 3.11.6 Protected

A task may be delete and/or execute protected. Protection is specified during the IT command and Install Task SVC.

**3.11.6.1 Delete Protected.** A delete-protected task cannot be deleted with the Delete Task (DT) command or the Delete Task SVC without removing protection with the Modify Task Entry (MTE) command. Delete protection is specified by the DELETE PROTECT? prompt during installation.

**3.11.6.2 Execute Protected.** Execute protection applies when a task contains only data and is not to be executed. Execute protection is specified by the Install Task SVC or by the EXECUTE PROTECT? prompt when using the IT command and applies to only the 990/12 computer.

**3.11.7 Copyable.** A task may be specified as copyable by responding to the IN MEMORY COPYABLE? prompt of the IT command or by a flag of the Install Task SVC. A copyable task may be copied from a copy in memory rather than from the disk copy.

**3.11.8 Reusable.** The IN MEMORY REUSABLE? prompt of the IT command or flag of the Install Task SVC specifies that after the task terminates, the task segment can be reused by another activation of the task. It does not have to be copied either from disk or memory.

**3.11.9 Updatable.** The UPDATABLE? prompt of the IT command or flag of the Install Task SVC specifies that the task segment on disk may be updated from the in-memory copy.

**3.11.10 Arithmetic Overflow Protection.** The 990/12 allows detection of overflow or underflow in arithmetic operations. This condition is signaled as a task error. Overflow detection is determined by the OVERFLOW CHECKING? prompt of the IT command or by a flag of the Install Task SVC.

**3.11.11 Writable Control Storage.** The 990/12 microcode resides in a special memory called the control store. The control store consist of the Read Only Memory control store (not discussed here) and the writable control store. The writable control store is composed of random-access memory (RAM) devices which contain user-written microcode. The writable control store is loaded by either DNOS or an assembly language instruction.

A portion of the writable control store and three Extended Operation (XOP) levels are reserved for Texas Instruments use. DNOS uses XOP levels 13, 14, and 15 and writable control store addresses >810 through >9FF to implement system routines. The user should not load microcode routines into these locations. Refer to the *Microcode Development System Programmer's Guide* for further information on control store management.

Use of the writable control storage is enabled or disabled by the WRITABLE CONTROL STORAGE? prompt of the IT command or by a flag of the Install Task SVC.

## 3.12 TASK TERMINATION

Tasks executing under DNOS may terminate normally or abnormally. In either case, the task should make provision for termination. If a task does not explicitly terminate using an End Task SVC, it either loops indefinitely or it attempts to exceed its memory bounds and causes abnormal termination. A task in an infinite loop may be killed externally by using the Kill Task (KT) or Kill Background Task (KBT) commands or by pressing first the RESET (blank orange) then the CONTROL X key when the task is running in the SCI foreground mode.

### 3.12.1 Normal Termination

To terminate normally, a task executes an End Task SVC. DNOS then releases the resources of the task and takes it out of execution. Disk-resident tasks are removed from memory. Memory-resident tasks remain in memory and occupy space but do not execute.

### 3.12.2 Abnormal Termination

If a task commits a fatal error (for example, illegal instruction, supervisor call, or memory reference), and the task entry vector includes an end action address, the routine at that address is activated. Typically, the end action routine executes an End Action Status SVC, outputs the returned data, and executes an End Task SVC. However, it is possible for the end action routine to implement error recovery procedures. If the end action routine commits errors or if the task includes no end action routine, DNOS unconditionally aborts the task and reports the error to the system log.

## 3.13 FILE AND DEVICE SERVICES

DNOS supports input and output operations to various types of devices, and to several types of files. In addition, DNOS supports communication between programs, in which each program is analogous to a peripheral device of the other. To include all types of I/O, this manual refers to devices, files, and communication channels between programs as I/O resources.

### 3.13.1 I/O Concepts

DNOS supports two concepts of I/O to resources. Many I/O operations apply to various devices and are essentially the same for each device. This concept is called resource-independent I/O. Resource-independent I/O allows the programmer to code I/O for terminals, magnetic tape units, line printers, card readers, and for sequential files, independently of the device or file.

A problem with resource-independent I/O is that operation of the resource is restricted to a mode that is common to other resources. Resource-specific I/O allows the programming of specific capabilities of the device. Resource-specific I/O supports terminals, special industrial devices, relative record files, key indexed files, and interprocess communication.

### 3.13.2 File and Device I/O

Several utility operations are required to support device I/O. A device may be specified by either a device name or by a logical name. The utility that manages logical names is used to assign a logical name. All I/O requires Logical Unit Numbers (LUNOs) which are assigned and deleted by another utility.

File I/O combines the support of the file management and I/O capabilities of DNOS for three types of disk files. The file types are:

- Sequential files
- Relative record files
- Key indexed files

Files may be accessed by a pathname or by a logical name, and a LUNO is required for I/O. The utility functions required for device I/O apply to file I/O also. In addition, functions exist to create a file, delete a file, verify or change the pathname, apply or change protection, and add or delete an alias.

# 4

# Building an Assembly Language Program

## 4.1 TEXT EDITOR USE

The Text Editor interactively creates and modifies files of textual data. The data in these files may be assembly language source code, high level language source code, or material that is to be printed, such as software documentation, memos, or drafts.

The user, working interactively, invokes and operates the Text Editor from a Model 911 Video Display Terminal (VDT). Most of the editing functions are available at both the VDT and hard copy terminals, but the means of invoking a particular function may vary depending on the terminal type and its current mode of operation. This manual assumes the use of a 911 VDT, being operated in VDT mode. Details about operation of a hard copy terminal can be found in the *DNOS Text Editor Reference Manual*.

The Text Editor is invoked by use of the Execute Text Editor (XE) command. The prompt for the filename allows the user to create a new file or edit an existing file.

To create a new file using the Text Editor, no pathname is entered when the editor is invoked. When the Text Editor is used to modify the data in an existing file, the user specifies the file name when the Text Editor is invoked. Each record in the input file is numbered, relative to the start of the file. The editor is terminated by use of the Quit Edit (QE) command. The user may abort the edit, by answering YES to the ABORT? prompt. In this case all modifications and new data are discarded. Answering NO to this prompt causes the edited file to be saved in the output file specified in response to the OUTPUT FILE ACCESS NAME prompt.

Errors detected by the Text Editor are described in the *DNOS Messages and Codes Reference Manual*.

## 4.2 TERMINAL USE

Text editing consists of four major types of operations:

- Command selection and specification
- Edit control
- Data display
- Data entry

Command selection and specification includes the selection of Text Editor functions that assist the user with the management of the text in the source file. The commands are listed in Table 4-1. Most of these commands have parameters that are supplied by the user, or, in many cases, by

default. After entering each parameter, press the TAB, RIGHT FIELD, or RETURN key to store the parameter. In addition, any System Command Interpreter (SCI) command may be called during text editing. The terminal must be in the command mode before selecting any command. The terminal is placed in the command mode by pressing the CMD key.

Edit control consists of the operations that control the immediate editing of data. The operations available are: altering the current file position, adding data by line, and deleting data by line, altering cursor position, adding data by character, and deleting data by character. Edit control operations have no parameters.

Data display manages the display of data on the device.

Data entry operations control the actual entering of data into the file.

Command selection from the 911 VDT is accomplished by keying in the command and responding to the prompts presented on the display screen. Edit control is performed by using the cursor control keys and some of the function keys.

Data entry or editing may occur on any record displayed on the screen by positioning the cursor anywhere within the line that contains the record to be edited. Records may be inserted between any lines, and may be inserted or deleted in any order. In addition, characters within a line may be inserted, deleted, or modified. The Show Line (SL) command, and the Roll Up, Roll Down, Cursor Up, and Cursor Down edit control functions listed in Table 4-2 position the file for display.

## 4.3 SCI COMMAND USE

The Text Editor is initiated when the user selects and completes the XE command and terminates when the user enters and completes the QE command. Whenever the terminal is in the command mode, the Text Editor is suspended and the user may select any SCI command. The commands selected when the terminal is in the command mode and the Text Editor is suspended are not necessarily Text Editor commands. The Text Editor remains suspended until the XE command or another Text Editor command is selected. Then the Text Editor is reactivated, the state that existed at the time of suspension is restored, and the entered command is processed. Any Text Editor command entered after the Text Editor has been terminated with the QE command causes the following message to be displayed:

```
U EDITOR-0009 COMMAND IS ALLOWED ONLY WHILE EDITING
```

If the user quits SCI (by entering the Quit (Q) command) while the editor is suspended, the QE command is automatically invoked, and the user must supply edit termination information.

Table 4-1 displays the valid commands of the Text Editor. For further information concerning the use of the Text Editor refer to the *DNOS Text Editor Reference Manual.*

**Table 4-1. Text Editor Commands**

| | |
|---|---|
| CL — Copy Lines | MT — Modify Tabs |
| DL — Delete Lines | QE — Quit Editor |
| DS — Delete String | RE — Recover Edit |
| FS — Find String | RS — Replace String |
| IF — Insert File | SL — Show Line |
| ML — Move Lines | SVL — Save Lines |
| MR — Modify Roll | XE — Execute Text Editor |
| MRM — Modify Right Margin | XES — Execute Text Editor with Scaling |

## 4.4 EDIT CONTROL FUNCTIONS

Edit Control functions are those that specify to the Text Editor precisely where within the file the modifications are to be made. Refer to Table 4-2 for the edit control functions supported and keys specified on the 911 VDT.

**Table 4-2. Edit Control Functions**

| Edit Control Function | 911 VDT Key Pressed |
|---|---|
| Command Mode | CMD |
| Roll (Display) Up | F1 |
| Roll (Display) Down | F2 |
| Duplicate To Tab | F4 |
| Clear To Tab | F5 |
| Display/Suppress Line Numbers [1] | F6 |
| Edit Mode/Compose Mode [2] | F7 |
| Begin Line | RETURN |
| Forward Tab [3] | TAB/SKIP (shifted) |
| Erase Rest of Line | TAB/SKIP |
| Forward Space One Character | CHAR → (shifted) |
| Backspace One Character | CHAR ← |
| Back Tab (Left Field) | FIELD ← |
| Right Margin (Right Field) | FIELD → (shifted) |
| Left Margin | ENTER |
| Erase Characters on Line | ERASE FIELD |
| Delete Line | ERASE INPUT |

**Table 4-2. Edit Control Functions (Continued)**

| Edit Control Function | 911 VDT Key Pressed |
|---|---|
| Insert Line | Blank Gray Key |
| Repeat (Input of Key) | REPEAT (plus key) |
| Insert Character | INS CHAR |
| Delete Character | DEL CHAR |
| Move Cursor Up | (Up Arrow) |
| Move Cursor Down | (Down Arrow) |
| Move Cursor Right | →(Right Arrow) |
| Move Cursor Left | ←(Left Arrow) |
| Move Cursor Home | HOME |
| Uppercase [2] | UPPERCASE LOCK |

**Notes:**

[1] Alternates display of line numbers (74 data characters) with no display of line numbers (80 data characters) each time key is pressed.

[2] Alternates modes each time the key is pressed.

[3] SHIFT key must be pressed concurrently with the TAB/SKIP key to achieve the tab function on the 911 VDT.

## 4.5 TEXT EDITOR EXAMPLE

The following paragraphs show examples of the Text Editor functions for creating a new file and for modifying an existing file. These examples provide a quick reference for the more common uses of the Text Editor. This simple program, RESPONSE, is also used in Section 9 to demonstrate the linking, installing, executing, and various debugging techniques discussed in this manual. Details of editor operations can be found in the *DNOS Text Editor Reference Manual.*

### 4.5.1 Creating a New File

The following procedure applies to creating a new file using the Text Editor on a 911 VDT in the VDT mode. The example assumes that you are properly logged-on, and that SCI is active. Refer to preceding section for details on log-on and activating SCI.

Create a directory .USER, on the system disk, with the Create File Directory (CFDIR) command. Enter CFDIR and press the RETURN key. Respond to the prompts as follows:

```
[ ] CFDIR

CREATE DIRECTORY FILE
                    PATHNAME:  .USER
                  MAX ENTRIES:  15
DEFAULT PHYSICAL RECORD SIZE:
```

Throughout the examples, files are automatically created in this directory.

Enter XE and press RETURN to activate the Text Editor. The following prompt appears:

```
[ ]XE

EXECUTE TEXT EDITOR
            FILE ACCESS NAME:
```

Press the RETURN key to indicate that no input file is to be edited. (Press the TAB SKIP key if there is a file pathname displayed.) The screen is cleared and the following display is presented in the first four columns of row one, with the cursor in column one, row one:

```
*EOF
```

This display indicates the only record in the file is the end-of-file record. To begin entering data, press the RETURN key to enter the first blank line into the file. The end-of-file record is now in line two and the cursor is in row one, column one, with the rest of the line blank. Enter data by positioning the cursor and keying the data. Press the RETURN key to return to column one and receive a new line. Enter the short assembly language program shown in Figure 4-1. Use the following column numbers to promote program readability.

1. The LABEL field begins in column 1
2. The OPERATION field begins in column 8
3. The OPERATOR field begins in column 13
4. The COMMENT field begins in column 26 or any column on a line with an asterisk (*) in column 1

Any of the edit control functions may be used during data entry, as may any SCI commands (press the CMD key to enter the command mode before entering an SCI command). Text editor commands return the Text Editor to the edit mode upon completion.

Once all the data has been entered, the Text Editor is terminated by calling the QE command. First, press the CMD key to enter the command mode, then enter QE and press the RETURN key. The following prompt appears:

```
[ ]QE

QUIT EDIT
                        ABORT?:  NO
```

```
*****************************************************************
*                          BEGINNING OF DATA SECTION
*****************************************************************
       IDT  'RESPONSE'
*****************************************************************
*      OPENING DATA WORDS
*          1. WORKSPACE POINTER
*          2. PC VALUE AT START OF PROGRAM
*          3. END ACTION ITEM (IF ANY)
*****************************************************************
*
       DATA WSP            WORKSPACE POINTER ADDRESS
       DATA START          PC AT PROGRAM BEGINNING
       DATA 0              END ACTION (NONE SPECIFIED)
WSP    BSS  32             WORKSPACE REGISTERS
OPEN   DATA 0              I/O REQUEST
       BYTE >0,>20         OPEN LUNO >20
       DATA 0
       DATA 0
       DATA 0
       DATA 0
MSSG0  DATA 0              I/O REQUEST
       BYTE >B,>20         WRITE ASCII TO LUNO >20
       DATA 0
       DATA GREET          MESSAGE LOCATION
       DATA 0
       DATA MSSG1-GREET    MESSAGE LENGTH
*
*****************************************************************
*      SPECIFY THE FIRST MESSAGE
*****************************************************************
*
GREET  DATA >0A0D
       TEXT 'HELLO, PLEASE INPUT NUMBER OF ITEMS '
       TEXT 'SOLD TODAY.  USE 4-DIGIT NUMBERS. '
       DATA >0A0D
```

**Figure 4-1. Assembly Language Program Example (Sheet 1 of 3)**

```
MSSG1   DATA 0              I/O REQUEST
        BYTE >B,>20         WRITE TO LUNO >20
        BYTE 0,>40          WRITE WITH REPLY
        DATA ITEM1
        DATA 0              CHARACTERS SPECIFIED IN INPUT ROUTE
        DATA 10             MESSAGE LENGTH
        DATA STR1           LOCATION OF INPUT PARAMETERS
STR1    DATA STORE          SAVE PARAMETERS IN STORE
        DATA 4              STORE FOUR CHARACTERS
        DATA 0
STORE   BSS  12
ITEM1   DATA >0A0D
        TEXT 'ITEM 1   '
MSSG2   DATA 0              I/O REQUEST
        BYTE >B,>20         WRITE TO LUNO >20
        BYTE 0,>40          WRITE WITH REPLY
        DATA ITEM2          MESSAGE LOCATION
        DATA 0
        DATA 10             MESSAGE LENGTH
        DATA STR2
STR2    DATA STORE+4        2ND ITEM CHARACTERS STORE LOCATION
        DATA 4
        DATA 0
ITEM2   DATA >0A0D
        TEXT 'ITEM 2   '
MSSG3   DATA 0              I/O REQUEST
        BYTE >B,>20         WRITE TO LUNO >20
        BYTE 0,>40          WRITE WITH REPLY
        DATA ITEM3          MESSAGE LOCATION
        DATA 0
        DATA 10             MESSAGE LENGTH
        DATA STR3
STR3    DATA STORE+6        3RD ITEM CHARACTERS STORE LOCATION
        DATA 4
        DATA 0
ITEM3   DATA >0A0D
        TEXT 'ITEM 3   '
MSSG4   DATA 0              I/O REQUEST
        BYTE >B,>20         WRITE TO LUNO >20
        DATA 0
        DATA GOODBY         MESSAGE LOCATION
        DATA 0
        DATA CLOSE-GOODBY MESSAGE LENGTH
```

**Figure 4-1. Assembly Language Program Example (Sheet 2 of 3)**

```
*
*********************************************************************
*       FINAL MESSAGE DISPLAYED
*********************************************************************
*
GOODBY  DATA >0A0D
        TEXT 'THANK YOU FOR YOUR PURCHASE. '
        TEXT ' HAVE A NICE DAY. '
        DATA >0A0D
CLOSE   DATA 0              I/O REQUEST
        BYTE 1,>20          CLOSE LUNO >20
        DATA 0
        DATA 0
        DATA 0
        DATA 0
EOP     BYTE >04,0          TERMINATE TASK
*
START   XOP  @OPEN,15       OPEN LUNO >20
        XOP  @MSSG0,15      OPENING MESSAGE
        XOP  @MSSG1,15      INPUT 1
        XOP  @MSSG2,15      INPUT 2
        XOP  @MSSG3,15      INPUT 3
        XOP  @MSSG4,15      EXIT MESSAGE
        XOP  @CLOSE,15      CLOSE FILE, UNLOAD/REWIND
        XOP  @EOP,15        TERMINATE TASK
        END  START
```

**Figure 4-1. Assembly Language Program Example (Sheet 3 of 3)**

The reply to the ABORT? prompt allows you to either accept (NO response) or discard (YES response) the data you entered. A YES response ignores all the data entered and leaves the file in its original form. Accept the data by pressing the RETURN key to accept the default value (NO). The example uses the NO response. The following display appears:

```
QUIT EDIT
   OUTPUT FILE ACCESS NAME:
                    REPLACE?:  NO
     MOD LIST ACCESS NAME:
```

The cursor appears after the colon in the first line of the display. Enter the pathname of the new output file, .USER.SOURCE and press the RETURN key. An entry is required since there is no input file.

Press the RETURN key to accept the NO default value of the REPLACE? prompt. The NO response allows you to avoid accidentally destroying an existing file.

Press the RETURN key in response to the MOD LIST ACCESS NAME prompt.

Once the file has been created, the Text Editor is no longer active and the terminal returns to command mode. The SCI prompt [ ] and a menu appear.

### 4.5.2 Editing an Existing File

The following example shows the general procedures for editing an existing file by using the Text Editor. The file used as input is the one shown in Figure 4-1.

First, press the CMD key to enter command mode. Enter XE and press the RETURN key. The following appears:

```
[ ]XE

EXECUTE TEXT EDITOR
            FILE ACCESS NAME:   .USER.SOURCE
```

The sample file, .USER.SOURCE, is displayed. (Enter .USER.SOURCE as the FILE ACCESS NAME response, if .USER.SOURCE is not displayed). Press the RETURN key to display the first 24-records from the file. The Text Editor is in the edit mode, the cursor is in column one, row one, and line numbers are displayed. In this example, we are going to modify the message, HELLO, PLEASE INPUT NUMBER OF ITEMS SOLD TODAY. USE 4-DIGIT NUMBERS. We are going to change HELLO to GOOD MORNING. To modify the message, perform the following:

1. Enter the command mode by pressing the CMD key.

2. Type RS to specify the Replace String command and press the RETURN key.

3. The prompts appear. Enter the following responses:

```
[ ] RS

REPLACE STRING
   NUMBER OF OCCURRENCES:  1
              START COLUMN:  14
                END COLUMN:  18
                    STRING:  HELLO
                    CHANGE:  GOOD MORNING
```

4. Press RETURN to activate the command processor.

5. When the Text Editor completes the string replacement, the line containing the old string, now changed, is displayed with the cursor in column one.

The RS command replaces the string HELLO with the string GOOD MORNING. Note that the occurrence number indicates that it replaces the first occurrence of the word HELLO that starts in column 14. Care should be taken that the occurrence number corresponds to the intended replacement. The advantage of using the RS command is that the text editing system finds the word in the file for you and replaces it without your having to manually delete the word and insert the correction.

Alternatively, the file can be edited manually. Once you have entered the file using the XE command, you can page through the text using the F1 function key to page forward, and the F2 function key to page backward. When you find the text you want to change, position the cursor over the word to be corrected. Use the DEL CHAR key to delete the word or characters. Then press the INS CHAR key and type in the characters to be inserted. Note that you can also locate a particular word in a file by using the FS command to find a specified string or word. The FS command operates very similarly to the RS command.

Once the modifications to the file are complete, press the CMD key to enter the command mode. Type QE and press the RETURN key. The following appears:

```
[ ]QE

QUIT EDIT
                          ABORT?:  NO
```

Press RETURN in response to the ABORT? prompt to save the modified file. The following appears:

```
QUIT EDIT
   OUTPUT FILE ACCESS NAME:  .USER.SOURCE
                  REPLACE?:  NO
      MOD LIST ACCESS NAME:
```

Press RETURN to accept .USER.SOURCE as the output file. Enter Y in response to the REPLACE? prompt and press the RETURN key twice. The Text Editor now terminates and the SCI prompt [ ] and menu return.

## 4.6 PROGRAMMING TECHNIQUES

The first step in developing an Assembly Language program is to clearly define the problem that the program is to solve and produce a good flowchart of the proposed solution. Try to break up the program into a series of subfunctions that can be coded in small modules: two to three hundred lines is a good limit for the size of a single module.

Determine what data you will need, both global and local to a subfunction, and how you will structure the data. Plan your subroutine linkage: calling, parameter passing, and returning protocols.

Data structures can be made global by using the External Reference (REF) and External Definition (DEF) directives or by putting them into a CSEG. If a CSEG is used, have only one source version of each CSEG. This version is copied into the modules requiring the CSEG information by the COPY directive.

Techniques that can be used for subroutine calls include using the Branch (B), Branch Indirect (BIND), Branch and Link (BL), Branch and Push Link to Stack (BLSK), or Branch and Load Workspace Pointer (BLWP) instruction. With the BL instruction, it is necessary to save the contents of register 11 before making a call from the called routine. This instruction is faster than the BLWP instruction and takes less memory. An advantage of using the BLWP instruction is that a fresh set of registers is available for use and the contents of the previously used set can be accessed by using register 13 as an index register. Status information can be passed back to the calling routine by placing data in bits 0-5 (status bits) of register 15. When the Return with Workspace Pointer (RTWP) instruction is executed, the contents of register 15 become the contents of the status register, and the calling routine can test the appropriate bits.

When coding individual modules, clearly identify the module using the Page Title (TITL) and Program Identifier (IDT) directives. Comments should be included at the beginning of the program stating the name and function of the module plus any useful information concerning parameters, calling conventions, and various programming methods used in coding the program. Instead of constants, use meaningful symbolic names defined with the Define Assembly-Time Constant (EQU) directive. Segmentation (use of the CSEG, DSEG, and PSEG directives) is mainly useful for coding shared procedures or RAM/ROM partitioned programs, but it can be used as documentation. Document the function or meaning of externally referenced symbols as well as locally defined symbols.

# 5

# Assembling a Program

## 5.1 OPERATING THE MACRO ASSEMBLER

The Macro Assembler is executed by the DNOS System Command Interpreter (SCI) in either background or batch mode.

To execute the Macro Assembler in background mode, enter the Execute Macro Assembler (XMA) command and press RETURN.

The XMA command prompts appear:

```
[ ] XMA

EXECUTE MACRO ASSEMBLER
          SOURCE ACCESS NAME:  pathname@          (*)
          OBJECT ACCESS NAME:  [pathname@]        (*)
          LISTING ACCESS NAME: pathname@          (*)
          ERROR ACCESS NAME:   [pathname@]        (*)
                     OPTIONS:  [characters]       (*)
     MACRO LIBRARY PATHNAME:   [pathname@]        (*)
          PRINT WIDTH (CHARS):  integer           (80)
          PAGE LENGTH (LINES):  integer           (60)
```

Enter the pathname or device name in response to the prompts described below. The Wait command may be entered after the last response.

```
[ ] WAIT

— WAITING FOR BACKGROUND TASK TO COMPLETE —
```

When the assembler terminates normally, the following message is displayed:

I ASSEMBLR-0001 MACRO ASSEMBLY COMPLETE, XXXX ERROR(S), YYYY WARNING(S)

The message displays the number of errors and warnings encountered, if any. Refer to Appendix B for the completion messages.

A description of the prompts follows:

SOURCE ACCESS NAME
: The input file or device containing assembly language code to be assembled.

OBJECT ACCESS NAME

The object code output file or device. If this parameter is null, no object output is produced. This is useful for preliminary assemblies to check for errors; since the assembler produces no output, it operates faster.

LISTING ACCESS NAME

The assembly listing file or device. If DUMY is entered, no assembly listing is produced.

ERROR ACCESS NAME

The assembly error output. This file may be viewed by entering the Show File (SF) command. If the ERROR ACCESS NAME is null, or if it is the same as the listing file, then errors are displayed on the terminal by the Show Background Status (SBS) command. If the device DUMY is specified, no error listing is produced.

The error file contains a complete list of any source records which cause assembly errors along with the other errors. If a condition is sensed which prevents the assembler from continuing, a message is written to the error file stating what has occurred. Then the user must enter the SBS command to view the error messages output by the assembler. Appendix A contains a list of abnormal completion messages and possible causes.

OPTIONS

Output and list options for the assembler. The user specifies any (or all) of the following options, separated by commas.

XREF — Prints a cross-reference listing at the end of the source and object listing file.

RXREF — Prints a reduced cross-reference listing at the end of the source and object listing file.

SYMT — Includes a symbol table with the output object code. This option must be specified to allow fully symbolic debugging.

TUNLST — Limits the listing for TEXT directives to a single line.

BUNLST — Limits the listing for BYTE directives to a single line.

DUNLST — Limits the listing for DATA directives to a single line.

MUNLST — Limits the listing for a macro expansion to a single line. TEXT, BYTE, and DATA statements and Macro usage often expand to produce multiple lines of code. If these options are selected, the statements appear in the listing but the expansion does not. For example, the source statement TEXT 'ABCDEF' produces the listing:

```
41 TEXT 'ABCDEF'
42
43
44
45
46
```

With the TUNLST option specified, only the line 41 TEXT 'ABCDEF' is produced in the listing.

FUNL — Overrides the unlist directives.

NOLIST — Suppresses all listing output, except to the error file. Overrides other directives and keywords.

12 — Specifies the full 990/12 instruction set. If the 990/12 instruction set is not specified, the Macro Assembler defaults to the 990/10 instruction set.

Any of the Option Key words may be abbreviated; for example, any of the following may be used for the TUNLST option:

```
T
TU
TUN
TUNL
TUNLS
TUNLST
```

To select more than one option, enter a list of keywords separated by commas. The keywords may appear in any order. To select all the options, one could enter the line:

```
OPTIONS: X,S,T,B,D,M
```

The options specified for this parameter are in addition to any options specified by OPTION directives in the source. (Refer to the *Assembly Language Reference Manual.*)

MACRO LIBRARY PATHNAME
: A directory containing macro definitions for this assembly. This pathname specification is equivalent to specifying the same pathname in a LIBIN directive, except that this pathname becomes the system macro library and is retained through stacked assemblies. This pathname is printed on the cover sheet of the first module only.

PRINT WIDTH (CHARS)
: Specifies the length of the lines to be written to the output file.

PAGE LENGTH (LINES)
: Specifies the number of lines per page in the listing file.

## 5.2 FORMAT OF GENERATED FILES

The assembler prints a source listing of the assembly code and the error or warning messages when these conditions are encountered. This section discusses the source listing, the error and warning codes output by the assembler, and the object code format.

### 5.2.1 Source Listing

The source listing shows the source statements and the resulting object code. A typical listing is shown in Figure 5-1.

```
SDSMAC
ACCESS NAMES TABLE                                          PAGE 0001

SOURCE ACCESS NAME=      .USER.SOURCE
OBJECT ACCESS NAME=      .USER.OBJECT
LISTING ACCESS NAME=     .USER.LISTING
ERROR ACCESS NAME=       .USER.ERROR
OPTIONS=
MACRO LIBRARY PATHNAME=

RESPONSE   SDSMAC
                                                            PAGE 0002
0001 0000
0002 0000
0003              ******************************************************
0004              *              BEGINNING OF DATA SECTION
0005              ******************************************************
0006                      IDT   'RESPONSE'
0007              ******************************************************
0008              *        OPENING DATA WORDS
0009              *           1.  WORKSPACE POINTER
0010              *           2.  PC VALUE AT START OF PROGRAM
0011              *           3.  END ACTION ITEM (IF ANY)
0012              ******************************************************
0013              *
0014 0000 0006'           DATA WSP       WORKSPACE POINTER ADDRESS
0015 0002 0142'           DATA START     PC AT PROGRAM BEGINNING
0016 0004 0000            DATA 0         END ACTION (NONE SPECIFIED)
0017 0006         WSP     BSS  32        WORKSPACE REGISTERS
0018 0026 0000    OPEN    DATA 0         I/O REQUEST
0019 0028   00            BYTE >0,>20    OPEN LUNO >20
     0029   20
0020 002A 0000            DATA 0
0021 002C 0000            DATA 0
0022 002E 0000            DATA 0
0023 0030 0000            DATA 0
0024 0032 0000    MSSG0   DATA 0         I/O REQUEST
0025 0034   0B            BYTE >B,>20    WRITE ASCII ON LUNO >20
     0035   20
0026 0036 0000            DATA 0
0027 0038 003E'           DATA GREET     MESSAGE LOCATION
0028 003A 0000            DATA 0
0029 003C 0050            DATA MSSG1-GREET     MESSAGE LENGTH
0030              *
0031              ******************************************************
0032              *        SPECIFY THE FIRST MESSAGE
0033              ******************************************************
0034              *
```

**Figure 5-1. Source Listing Example**

The assembler produces a cover page as the first output in the listing. This cover page contains a table that provides a record of the files and devices used during the assembly process. Figure 5-2 is an example of this output when a macro library is specified. A macro library is a directory containing files of macro definitions. The assembler directives LIBIN and LIBOUT identify macro libraries where macro definitions are read and written.

The output has two sections:

- A listing of the parameters that were passed to the assembler via SCI.
- A list of access names encountered during the first pass of the assembly.

In the first section, any parameters that have no values are left blank. The fields in the second section are labeled as follows:

LINE — This field contains the record number in which the access name was encountered.

KEY — This field contains one of the following:

LI (indicating a LIBIN usage)

LO (indicating a LIBOUT usage)

single character (a single character key assigned to a copy file)

```
SDSMAC

ACCESS NAMES TABLE                                          PAGE 0001

SOURCE ACCESS NAME=              . USER. SRC. TEST1
OBJECT ACCESS NAME=
LISTING ACCERSS NAME=            . USER. LIST. TEST1
ERROR ACCESS NAME=
OPTIONS=                         XREF, SYMT, TUNLST, MUNLST
MACRO LIBRARY PATHNAME=          . SDSMAC. MACRODEF

LINE      KEY        NAME

0001      LI         . SDSMAC. MACRODEF
                       =>. SDSMAC. MACRODEF
0001      LO         MACROS
                       =>. SDSMAC. MACRODEF
0002       A         DSC. SYSTEM. TABLES. DDR
                       =>DS01. SYSTEM. TABLES. DDR
0003      LI         . SDSMAC. MACRODEF
                       =>. SDSMAC. MACRODEF
```

**Figure 5-2. Output Cover Page Example**

NAME — This field contains two access names. The first name is the name that appears in the source record. The second name, following the =>, is the result of synonym substitution in the first name.

Each page of the source listing has a title line at the top of the page. Any title supplied by a TITL directive is printed on this line, and a page number is printed to the right of the title area. The printer skips a line below the title line, and prints a line for each source statement listed. The line for each source statement contains a source statement number, a location counter value, object code assembled, and the source statement as entered. When a source statement results in more than one word of object code, the assembler prints the location counter value and object code on a separate line succeeding the source statement. The source listing lines for some example source statements are shown in Figure 5-3:

The source statement number, 0014 in the example, is a four-digit decimal number. Source records are numbered in the order in which they are entered, whether they are listed or not. The TITL, LIST, UNL, and PAGE directives are not listed, and source records between a UNL directive and a LIST directive are not listed. The difference between source record numbers printed indicates how many source records are not listed.

The next field on a line of the listing contains the location counter value, a hexadecimal value. In the example, 0000 is the location counter value. Not all directives affect the location counter, and those that do not affect the location counter leave this field blank. Specifically, the directives IDT, REF, DEF, DXOP, EQU, SREF, LOAD, and END leave the location counter field blank.

The third field normally contains a single blank. However, the assembler places a dash in this field when warnings are detected.

The fourth field contains the hexadecimal representation of the object code placed in the location by the assembler, 0006 in the example. The apostrophe following the fourth field indicates that the content, 0006, is program-relocatable. A relocatable address is an address which is relative to the base address of a particular code segment. The link editor will modify the address by adding the base address of the segment to it when the base address is determined. A program-relocatable address is an address relative to the beginning of a program segment (PSEG). A quote (') in this location indicates the location is data-relocatable, the address is relative to the base address of a data segment (DSEG). A plus (+) indicates the label is relocatable with respect to a common segment (CSEG). All machine instructions and the BYTE, DATA, and TEXT directive use this field for object code. The EQU directive places the value corresponding to the label in the object code field.

```
0014 0000 0006'         DATA WSP        WORKSPACE POINTER ADDRESS
0015 0002 0142'         DATA START      PC AT PROGRAM BEGINNING
0016 0004 0000          DATA 0          END ACTION (NONE SPECIFIED)
0017 0006         WSP   BSS  32         WORKSPACE REGISTERS
```

**Figure 5-3. Source Statement Listing Example**

The fifth field contains the first n-characters of source statement as supplied to the assembler, where n equals a print width between 20 and 80, inclusive. Spacing in this field is determined by the spacing in the source statement. The four fields of source statements will be aligned in the listing only when they are aligned in the same character positions in the source statements or when tab characters are used.

### 5.2.2 Error Messages

The assembler prints the following error message on successive lines of the listing when an error is detected:

```
*** error description ***

LAST ERROR AT XXXX
```

The second line identifies the statement in which the previous error was detected.

At the end of the listing is an error summary, as follows:

```
NNNN ERRORS, LAST ERROR AT XXXX,YYYY WARNINGS
```

NNNN is the count of errors in the assembly. XXXX identifies the last error detected in the assembly. YYYY is the count of the warnings in the assembly. The second lines of the error messages link the error messages so that the user may begin at the error summary message and readily locate all error messages. In an error-free assembly, the final message is:

```
NO ERRORS, NO WARNINGS or NO ERRORS, XXXX WARNINGS
```

Several errors detected by the assembler (such as arithmetic overflow while evaluating expressions) are considered to be only warnings. The programmer should examine the code generated when warning messages occur since the results may or may not be the code expected. Warning messages are written only to the error file and are not included in the listing. However, a dash is placed in column 11 of the listing where the warning error occurred. Warning messages do not include an indication of a previous warning or error. Refer to the Appendix C for listing error messages.

### 5.2.3 Cross-Reference Listing

The assembler prints an optional cross-reference listing following the source listing. The format of the listing is shown in Figure 5-4. In the LABEL column, the assembler prints each symbol defined or referenced in the assembly program. In the second column, the attributes of the symbol are indicated as a list of single characters. The characters that appear in the second column, and their meanings, are listed in Table 5-1. The VALUE column contains a four-digit hexadecimal number assigned to the symbol. The DEFN column is the statement number of the defined symbol. For undefined symbols, the fourth column is left blank. The REFERENCE column contains a list of the statement numbers that reference the symbol. This column is left blank if the symbol is unused. An apostrophe in the value column means that the content is program-relocatable.

```
                                CROSS REFERENCE

LABEL                VALUE      DEFN       REFERENCES

ADDT                 01A8'      0325       0314
ADSR       D         01A0'      0316       0342      0343      0348      0349
GT                   0006       0997
```

**Figure 5-4. Cross Reference Listing**

Table 5-1 shows the characters that appear in the second column of the cross-reference and their meanings.

**Table 5-1. Symbol Attributes**

| Character | Meaning |
|---|---|
| R | External reference (REF) |
| D | External definition (DEF) |
| U | Undefined |
| M | Macro name |
| S | Secondary reference (SREF) |
| L | Force load (LOAD) |

### 5.2.4 Object Code

The assembler produces object code to load into the 990 computer. This object code may be linked to other object code modules (or programs) or loaded directly. Object code consists of records containing up to 71 ASCII characters each. The format, described in the next paragraph, permits corrections of errors without reassembling the program (discussed in Procedures on Changing Object Code paragraph). An example of output object code is shown in Figure 5-5.

**5.2.4.1 Object Code Format.** The object record consists of a number of tag characters, each followed by one to three fields as defined in Table 5-2. The first character of a record is the first tag character, which tells the loader which field or fields follow the tag. The next tag character follows the end of the field or fields associated with the preceding tag character. When the assembler has no more data for the record, the assembler writes tag character 7 followed by its field in turn followed by the tag character F. The assembler then fills the rest of the record with blanks and a sequence number, and begins a new record with the appropriate tag character.

```
0015CRESPONSEA0000C0006C013CB0000A0006A0026B0000B0020B0000B00007F211F    RESP0001
B0000B0000B0000B0B20B0000C003EB0000B004AB0A0DB4845B4C4CB4F2CB20507F1CFF RESP0002
B4C45B4153B4520B494EB5055B5420B4E55B4D42B4552B204FB4620B4954B454D7F187F RESP0003
B5320B534FB4C44B2054B4F44B4159B2E20B2055B5345B2034B2D44B4947B49547F199F RESP0004
B204EB554DB4245B5253B2E00A0086B0A0DB0000B0B20B0040C00A8B0000B000A7F1C8F RESP0005
C0096C009CB0004B0000A009CA00A8B0A0DB4954B454DB2031B2020B0000B0B207F1D1F RESP0006
B0040C00C6B0000B000AC00C0C00A0B0004B0000B0A0DB4954B454DB2032B20207F1EFF RESP0007
B0000B0B20B0040C00E4B0000B000ACC0DEC00A2B0004B0000B0A0DB4954B454D7F1CEF RESP0008
B2033B2020B2000A00F0B0000B0B20B0000C00FCB0000B0032B0A0DB5448B414E7F1F1F RESP0009
B4B20B594FB5520B464FB5220B594FB5552B2050B5552B4348B4153B452EB20487F197F RESP0010
B4156B4520B4120B4E49B4345B2044B4159B2E00A012CB0A0DB0000B0120B00007F1E6F RESP0011
B0000B0000B0000B0400B2FE0C0026B2FE0C0032B2FE0C0088B2FE0C00B2B2FE07F195F RESP0012
C00D0B2FE0C00F0B2FE0C012EB2FE0C013A7F7D9F                                RESP0013
2013C7FEC0F                                                              RESP0014
:       RESPONSE                                                         RESP0015
```

**Figure 5-5. Object Code Example**

**Table 5-2. Object Record Format and Tags**

| Tag | Field 1 | Field 2 | Field 3 |
|---|---|---|---|
| MODULE DEFINITION | | | |
| 0 | PSEG Length | Program ID (8) | — |
| M | DSEG Length | $DATA | 0000 |
| M | Blank Common Length | $BLANK | Common # |
| M | CSEG Length | Common Name (6) | Common # |
| M | CBSEG Length | $CBSEG | CBSEG # * |
| ENTRY POINT DEFINITION | | | |
| 1 | Absolute Address | — | — |
| 2 | P-R Address | — | — |
| LOAD ADDRESS | | | |
| 9 | Absolute Address | — | — |
| A | P-R Address | — | — |
| S | D-R Address | — | — |
| P | C-R Address | Common or CBSEG # | — |
| DATA | | | |
| B | Absolute Value | — | — |
| C | P-R Address | — | — |
| T | D-S Address | — | — |
| N | C-R Address | Common or CBSEG # | — |
| EXTERNAL DEFINITIONS | | | |
| 5 | P-R Address | Symbol (6) | — |
| 6 | Absolute Value | Symbol (6) | — |
| W | D-R/C-R Address | Symbol (6) | Common # |

**Table 5-2. Object Record Format and Tags (Continued)**

| Tag | Field 1 | Field 2 | Field 3 |
|---|---|---|---|
| EXTERNAL REFERENCES | | | |
| 3 | P-R Address of Chain | Symbol (6) | — |
| 4 | Absolute Address of Chain | Symbol (6) | — |
| X | D-R/C-R Address of Chain | Symbol (6) | Common # |
| E | Reference Index Number | Absolute Offset | — |
| SYMBOL DEFINITIONS | | | |
| G | P-R Address | Symbol (6) | — |
| H | Absolute Value | Symbol (6) | — |
| J | D-R/C-R Address | Symbol (6) | Common # |
| FORCE EXTERNAL LINK | | | |
| U | 0000 | Symbol (6) | — |
| SECONDARY EXTERNAL REFERENCE | | | |
| V | P-R Address of Chain | Symbol (6) | — |
| Y | Absolute Address of Chain | Symbol (6) | — |
| Z | D-R/C-R Address of Chain | Symbol (6) | Common # |
| CHECK SUM | | | |
| 7 | Value | — | — |
| IGNORE CHECK SUM * | | | |
| 8 | Any Value | — | — |
| LOAD BIAS * | | | |
| D | Absolute Address | — | — |
| END OF RECORD | | | |
| F | — | — | — |
| REPEAT COUNT (FORTRAN OPTION) * | | | |
| R | Value | Repeat Count | — |
| PROGRAM ID (SYMT OPTION) * | | | |
| i | P-R Address | Program ID (8) | — |
| COBOL SEGMENT REFERENCE * | | | |
| Q | Record Offset | CBSEG # | — |

**Notes:**

All field widths are four-characters unless otherwise specified by numbers in parentheses

If the first tag is >01, the file is in compressed object format

P-R Program Segment Relative (address)
D-R Data Segment Relative (address)
C-R Common Segment Relative (address)

* Not generated during assembly.

Tag Character 0

This tag character is followed by two fields.

Field 1 — The number of bytes of program-relocatable code.

Field 2 — The program identifier assigned to the program by an IDT directive. When no IDT directive is entered, the field contains blanks. The linker uses the program identifier to name the program. The number of bytes of program-relocatable code determines the load bias for the next module or program. The assembler places a single tag character 0 at the beginning of each program.

Tag Character M

This tag character is used when data or common segments are defined in the program and is followed by three fields. COBOL also uses this tag for special code segments.

Field 1 — The length, in bytes, of data- or common-relocatable code.

Field 2 — The data or common segment identifier. The identifier is a six-character field containing the name $DATA (there must be a blank after $DATA) for data segments and $BLANK for blank common segments. If a named common segment appears in the program, an M tag will appear in the object code with an identifier field corresponding to the operand in the defining CSEG directive(s). The name $CBSEG is used for COBOL special code segments.

Field 3 — A four-character hexadecimal number defining a unique "common number" to be used by other tags which reference or initialize data of that particular segment. For data segments, this common number is always zero. For common segments (including blank common), a common number is assigned to each segment beginning with one and ending with the number of different common segments. The maximum number of common segments that a program may contain is 126.

Tag Characters 1 and 2

These tag characters are used with entry addresses.

1 — Used when the entry address is absolute.

2 — Used when the entry address is program-relocatable.

Field 1 — The entry address in hexadecimal. One of these tags may appear at the end of the object code file. The associated field is used by the loader to determine the entry point at which execution starts when the loading is complete.

Tag Characters 9, A, S, and P

These tag characters are used with load addresses for data that follows. Tag P contains two fields, all the other tags contain only one field.

9 — Used when the load address is absolute.

A — Used when the load address is program-relocatable.

S — Used when the load address is data-relocatable.

P — Used when the load address is common-relocatable.

Field 1 — The address at which the following data word is to be loaded. A load address is required for a data word that is to be placed in memory at some address other than the next address. The load address is used by the loader.

Field 2 — The common number for tag character P.

Tag Characters B, C, T, and N

These tag characters are used with data words. Tag N contains two fields; all the other tags contain only one field.

B — Used when the data is absolute (an instruction word or a word that contains text characters or absolute constants).

C — Used for a word that contains a program-relocatable address.

T — Used for a word that contains a data-relocatable address.

N — Used for a word that contains a common-relocatable address.

Field 1 — The data word. The loader places the data word in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word.

Field 2 — The common number for tag character N.

Tag Characters 5, 6, and W

These tag characters are used for external definitions. Tag W consists of three fields; the other two tags contain two fields.

5 — Used when the location is program-relocatable.

6 — Used when the location is absolute.

W — Used when the location is data- or common-relocatable.

The fields are used by the Link Editor to provide the desired linking to the external definition.

Field 1 — The defined value of the external symbol.

Field 2 — The symbol which is being defined.

Field 3 — The common number for tag character W.

Tag Characters 3, 4, and X

These tag characters are used for external references. Fields 1 and 2 are used by the linker to provide the desired linking to the external reference.

3 — Used when the last appearance of the symbol in field 2 of the tag is in program-relocatable code.

4 — Used when the last appearance of the symbol is in absolute code.

X — Used when the last appearance of the symbol is in data- or common-relocatable code.

Field 1 — The location of the last appearance of the symbol.

- When the location of the last appearance is absolute zero, no location in the program requires the address corresponding to the reference.

- When the location of the last appearance is (__not) absolute zero, that location serves as the base address of a back chain. The various locations of uses of an external reference are chained together with each link in the chain pointing to a location which has appeared previously in the object code. Thus the contents of the base address of the back chain is the address of the immediately preceding link in the chain. The location of the final link will contain absolute zero.

Field 2 — The external reference.

Field 3 — Tag character X, which gives the common number.

For each external reference in a program, there is a tag character in the object code with a location or an absolute zero, and the symbol referenced.

Figure 5-6 illustrates the chain of the external reference (EXTR).

The object code contains the following tag and fields:

4C00EEXTR

At location C00E, the address C00A points to the preceding appearance of the reference. The chain includes both absolute and relocatable addresses. The absolute addresses are C00E, C00A, C006, C002, B00E, B00A, B006, and B002. The relocatable addresses are 029E, 029A, 0298, 0294, 0290, and 028E. Each location points to the preceding appearance, except for location 028E, which contains zero. The zero identifies location 028E as the first appearance of EXTR, the end of the chain.

Tag Character E

This tag character is also used for external references. An E tag is used when a non-zero quantity is to be added to a reference.

Field 1 — The index into references identified by 3, 4, V, X, Y, and Z tags in the object code.

```
0229                    *          DEMONSTRATE EXTERNAL
0230                    *          REFERENCELINKING
0231                    *
0232                               REF  EXTR
0233  028C                         RORG
0234  028C  C820                  MOV  @EXTR,@EXTR
      028E  0000
      0290  028E'
0235  0292  28E0                  XOR  @EXTR,3
      0294  0290'
      0236  B000                        AORG >B000
0237  B000  3220                  LDCR @EXTR,8
      B002  0294'
0238  B004  0420                  BLWP @EXTR
      B006  B002
0239  B008  0223                  AI   3,EXTR
      B00A  B006
0240  B00C  38A0                  MPY  @EXTR,2
      B00E  B00A
0241  0296                         RORG
0242  0296  C820                  MOV  @EXTR,@EXTR
      0298  B00E
      029A  0298'
0243  029C  28E0                  XOR  @EXTR,3
      029E  029A'
0244  C000                        AORG >C000
0245  C000  3220                  LDCR @EXTR,8
      C002  029E'
0246  C004  0420                  BLWP @EXTR
      C006  C002
0247  C008  0223                  AI   3,EXTR
      C00A  C006
0248  C00C  38A0                  MPY  @EXTR,2
      C00E  C00A
```

**Figure 5-6. External Reference**

Field 2 — The value to be added to the reference after the reference is resolved.

The list is maintained by order of occurrence (the first entry in the list is the symbol located in field 2 of the first 3, 4, V, X, Y, or Z tag.) The index to that reference in the E tag would be 0000.

Tag Characters G, H, and J

These tag characters are used when the symbol table option is specified with the assembler. Tag J contains three fields, all the other tags contain two fields.

G — Used when the location or value of the symbol is program-relocatable.

H — Used when the location or value of the symbol is absolute.

J — Used when the location or value of the symbol is data- or common-relocatable.

Field 1 — The location or value of the symbol.

Field 2 — The symbol to which the location is assigned.

Field 3 — The common number for tag character J.

Tag Character U

This tag character is generated by the LOAD directive. The symbol specified is treated as if it were the value specified in an INCLUDE command to the loader.

Field 1 — All zeros.

Field 2 — The symbol to be defined. Refer to the LOAD directive in the *Assembly Language Reference Manual* for further information.

Tag Characters V, Y, and Z

These tag characters are used for secondary external references. The three fields are used by the Link Editor to provide linking to the secondary external reference.

V — Used when the last appearance of the symbol is in program-relocatable code.

Y — Used when the last appearance of the symbol is in absolute code.

Z — Used when the last appearance of the symbol is in data- or common-relocatable code.

Field 1 — The location of the last appearance of the symbol.

Field 2 — The symbol that is used as the secondary external reference.

Field 3 — The common number for tag character Z.

Tag Character 7

This tag character precedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the two's complement of the summed 8-bit ASCII character values from the first tag through tag 7.

Field 1 — The checksum value.

Tag Character 8

This tag character is used to ignore the checksum and is not generated at assembly.

Field 1 — The checksum value to be ignored.

Tag Character D
This tag character is used to specify a load bias and is not generated during assembly.

Field 1 — The absolute address used by the loader to relocate the symbols when loading. The Link Editor does not accept the D tag.

Tag Character F
This tag character indicates the end of record. It may be followed by blanks.

Tag Character R
This tag character is generated during FORTRAN compilation and represents the repeated count of an absolute value (B tags).

Tag Character I
This tag character represents the base address of a module and is generated by the Link Editor.

Field 1 — The base address of the named module in the linked object.

Field 2 — IDT name of the module.

Tag Character Q
This tag character is generated during COBOL compilation. This tag is the segment identifier to the overlay directory entry.

Field 1 — The record offset.

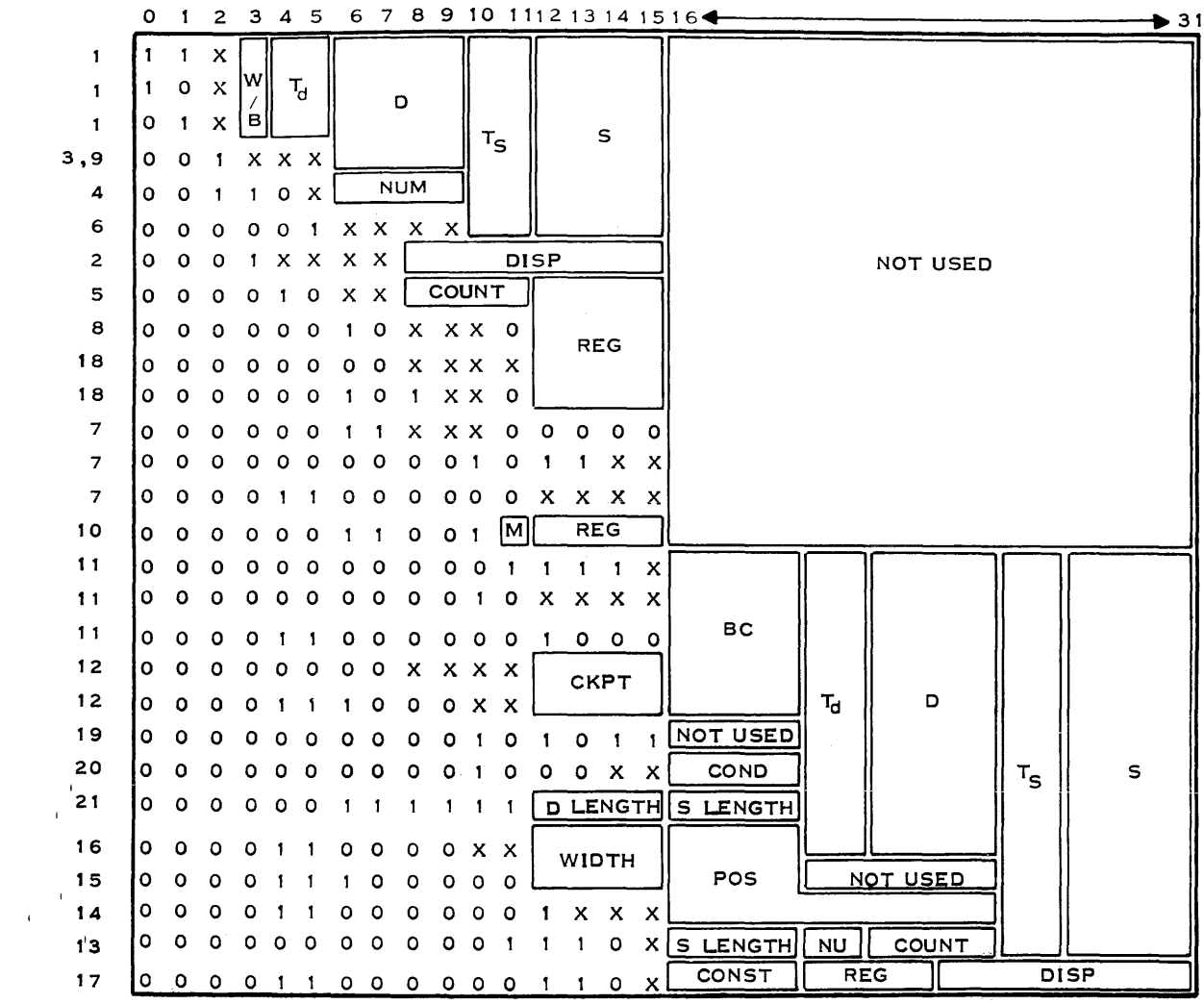
Field 2 — CBSEG number.

The last record of an object module has a colon (:) in the first character position of the record, followed by blanks or a time and date identifying stamp.

**5.2.4.2 Machine Language Format.** Some of the data words preceded by tag character B represent machine instructions. Comparing the source listing with the object code fields identifies the data words that represent machine instructions. Figure 5-7 shows the manner in which the bits of the machine instructions relate to the operands in the source statements for each format of machine instruction.

**5.2.4.3 Symbol Table.** When the SYMT option is specified, the symbol table is included in the object code file. One entry, using tag character G or H as appropriate, is supplied for each symbol defined in the assembly.

**5.2.4.4 Procedures for Changing Object Code.** In most cases, changing the object code to correct errors in a program is not recommended. All changes or corrections to a program should be made in the source code. Then the program should be reassembled. Failure to follow this principle can make subsequent correction or maintenance of the program impossible. The information in the following paragraphs is intended for those rare instances when reassembly is not possible. Any changes made directly to the object code should be thoroughly documented to show what the program actually does, not what the source code says it does. To correct the object code without reassembling a program, change the object code by changing or adding one or more




| | |
|---|---|
| X | IS A BIT OF THE OPERATION CODE THAT IS EITHER 0 OR 1 ACCORDING TO THE SPECIFIC INSTRUCTION IN THE FORMAT |
| W/B | IS A BIT IN THE OPERATION CODE THAT IS 0 IN INSTRUCTIONS THAT OPERATE ON WORDS, AND 1 IN INSTRUCTIONS THAT OPERATE ON BYTES |
| $T_D$ | IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE DESTINATION OPERAND, AS FOLLOWS:<br>00 = WORKSPACE REGISTER ADDRESSING<br>01 = WORKSPACE REGISTER INDIRECT ADDRESSING<br>10 = SYMBOLIC MEMORY ADDRESSING WHEN D = 0<br>10 = INDEXED MEMORY ADDRESSING WHEN D ≠ 0<br>11 = WORKSPACE REGISTER INDIRECT AUTOINCREMENT ADDRESSING |
| D | IS THE WORKSPACE REGISTER FOR THE DESTINATION OPERAND |
| $T_S$ | IS A PAIR OF BITS THAT SPECIFY THE ADDRESSING MODE OF THE SOURCE OPERAND AS SHOWN FOR $T_D$ |
| S | IS THE WORKSPACE REGISTER FOR THE SOURCE OPERAND |
| NUM | IS THE NUMBER OF BITS TO BE TRANSFERRED |
| DISP | IS A TWO'S COMPLEMENT NUMBER THAT REPRESENTS A DISPLACEMENT |
| REG | IS A WORKSPACE REGISTER ADDRESS |
| COUNT | IS A SHIFT COUNT |
| M | IS A MAP REGISTER FILE NUMBER (0 OR 1) |
| BC | IS A BYTE COUNT |
| CKPT | IS A CHECKPOINT REGISTER ADDRESS |
| COND | IS A LOGICAL SEARCH CONDITION (EQ,GT,ETC.) |
| D LENGTH | IS A BYTE COUNT OF THE DESTINATION OPERAND |
| S LENGTH | IS A BYTE COUNT OF THE SOURCE OPERAND |
| WIDTH | IS THE NUMBER OF BITS CONTAINED IN THE OPERAND |
| POS | IS A BIT POSITION |
| CONST | IS A CONSTANT TO BE ADDED TO OR SUBTRACTED FROM A WORKSPACE REGISTER |
| NOT USED | IS A GROUP OF BITS NOT USED IN THE INSTRUCTION |
| N U | NOT USED |

2279942

**Figure 5-7. Machine Instruction Formats**

records. One additional tag character is recognized by the loader to permit specifying a load point. The additional tag character D, may be used in object records changed or added manually.

Tag character D is followed by a load bias (offset) value. The loader uses this value instead of the load bias computed by the loader itself. The loader adds the load bias to all relocatable entry addresses, external references, external definitions, load addresses, and data. The effect of the D tag is to specify the area of memory for loading. The tag character D and the associated field must precede the assembler generated object code.

**NOTE**

Both linked object code and object code loaded by the boot loader can be changed without reassembling the program. However, the Link Editor will not accept tag character D in changed or added object records.

Correction of the object code may require only changing a character or a word in an object code record. The user may duplicate the record up to the character or word in error, replace the incorrect data with the correct data, and duplicate the remainder of the record up to the tag character 7. Because the changes the user has made will cause a checksum error when the checksum is verified as the record is loaded, the user must change tag 7 to tag 8.

When more extensive changes are required, the user may write an additional object code record or records. Begin each record with a tag character 9, A, S, or P, followed by an absolute load address or a relocatable load address. This may be an address into which an existing object code record places a different value. The new value on the new record will override the other value when the new record follows the other record in the loading sequence. Follow the load address with a tag character B, C, or N, and an absolute data word or a relocatable data word. Additional data words preceded by appropriate tag characters may follow. When additional data is to be placed at a non-sequential address, write another load address tag character (9, A, S, or P), the load address and data words, and precede with the appropriate tag character (B, C, or N). When the record is full, or all changes have been written, write tag character F to end the record.

When additional memory locations are loaded as a result of changes, the user must change field 1 of tag character 0 which contains the number of bytes of relocatable code. For example, when the object file written by the assembler contains >1000 bytes of relocatable code, and the user adds 8 bytes in a new object record, additional memory locations will be loaded. Therefore, the user must find the first 0 tag character in the object code file and change the value following the tag value from 1000 to 1008. The 7 tag character must be changed to a tag 8 in that record.

When added records place corrected data in locations previously loaded, the added records must follow the incorrect records. The loader processes the records as they are read from the object medium. The last record that affects a given memory location determines the contents of that location at execution time.

The object code records that contain the external definition fields, the external reference fields, the entry address field, and the final program start field must follow all other object records. An additional field or record may be added to include reference to a program identifier with the tag character 4. In this case, Field 1 contains 0 and Field 2 contains the first six characters of the IDT

character string. External definitions may be added using tag character 5 or 6 followed by the relocatable or absolute address, respectively, in Field 1. Field 2 contains the first six characters of the defined symbol, left justified and blank filled to the right.

## 5.3 OPERATING THE ASSEMBLER IN BATCH MODE

Operating the Macro Assembler in batch mode requires two steps:

1. Prepare the batch command stream.

2. Execute the batch command stream using the Execute Batch (XB) or the Execute Batch Job (XBJ) command.

Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for information related to batch mode execution and batch stream examples.

### 5.3.1 Batch Stream Structure

The Batch command stream for macro assembly is depicted in Figure 5-8.

Any sequential media (cards, cassette, magnetic tape, or sequential file) may be used for the batch stream.

The parameters for records in a Macro Assembly batch stream are the following:

BATCH and EBATCH
: In order to remove unwanted synonyms and default values, the BATCH command should be the first command in any batch stream and the EBATCH command should be the last command.

XMA record
: Specifies the Macro Assembly and supplies the required parameters. Parameters are supplied in the following format:

    field prompt = value

The prompts assign the first, second, and other parameters associated with the command. A prompt is either the full field prompt associated with each parameter, or an abbreviation that includes enough characters to identify the field prompt. Often, only the first character of a field prompt has to be entered. For example, to specify the source file .ALFILE, the following characters may be used:

```
SOURCE = .ALFILE

         or

S = .ALFILE
```

```
BATCH
XMA  S=.ALFILE, O=.ALOBJ, L=.ALLIST
EBATCH
```

**Figure 5-8. Macro Assembly Batch Stream**

When a prompt takes a list as input, the list must be enclosed in parentheses:

OPTIONS = (X,T,U)

Each prompt response must be separated from other responses by a comma. For example, the following record assembles a source file named .SOURCE, producing an object file (.OBJECT), a listing file (.LIST), and reporting errors to .ERR; the options selected are cross reference (XREF) and symbol table (SYMT); no macro library is to be used:

XMA S=.SOURCE,OB=.OBJECT,L=.LIST,E=.ERR,OP=(X,S)

The only required parameters are SOURCE and LISTING. Other parameters may use initial values as indicated in the paragraph on background processing.

### 5.3.2 Execute Batch Command

To execute a batch stream, enter the Execute Batch (XB) command and press RETURN. The following appears:

*Prompts:*

```
EXECUTE BATCH
          INPUT ACCESS NAME:  pathname@
      LISTING ACCESS NAME:  pathname@
```

*Prompt Details:*

INPUT ACCESS NAME:
The pathname from which SCI should read the batch command stream.

LISTING ACCESS NAME:
The pathname of a device or file to which SCI should write the results of the batch command stream execution. This device or file must not be used by any command in the batch command stream.

*Example:*
In the following example, the XB command will execute a batch stream from a file and output the results of the batch command stream to a line printer.

```
[] XB
EXECUTE BATCH
          INPUT ACCESS NAME:  MY.BATCH
      LISTING ACCESS NAME:  LP01
```

### 5.3.3 Execute Batch Job

The Execute Batch Job (XBJ) command allows a user to create a batch job with different operating parameters than those of the creating job.

To execute the XBJ command, enter XBJ and press RETURN. The following appears:

*Prompts:*

```
EXECUTE BATCH SCI JOB
                        JOB NAME:  alphanumeric
                USE CURRENT USER ID?:  YES/NO          (YES)
LOGICAL NAME TABLE PATHNAME:  [filename@]
          SYNONYM TABLE PATHNAME:  [filename@]
```

If NO is the response to the USE CURRENT USER ID? prompt, the following prompts are displayed:

```
EXECUTE BATCH SCI JOB
                          USER ID:  alphanumeric
                       PASSCODE:  [characters]
                     ACCOUNT ID:  [characters]
```

If the user's response to the SYNONYM TABLE PATHNAME prompt is null, the following prompts are displayed on the user's terminal:

```
                  INPUT ACCESS NAME:  pathname@
                 LISTING ACCESS NAME:  pathname@
```

*Prompt Details:*

JOB NAME:
A one- to eight-character, user-defined string by which the user wishes to reference the job.

USE CURRENT USER ID:
If YES is specified, the current user ID is used. If the response is NO, the user must specify a new user ID.

USER ID:
The user ID to be associated with the new job. A response to this prompt is required if the response to the USE CURRENT USER ID? prompt was NO.

PASSCODE:
The passcode corresponding to the user ID of the new job.

ACCOUNT ID:
A 1- to 16-character string that is the account ID for the new job.

LOGICAL NAME TABLE PATHNAME:
The file name containing the logical names to be passed to the new job. A null response passes the creating job's logical names. Any other entry is considered a file name containing logical names established by the Snapshot Name Definition (SND) command. The default for this prompt is a null value.

SYNONYM TABLE PATHNAME:
The file name containing the set of synonyms to be used by the new job. The file must have been created via the SND command and must include the new job's input and listing access names in its parameter list. If a null response is entered, the INPUT ACCESS NAME and LISTING ACCESS NAME prompts are displayed at the user's terminal.

INPUT ACCESS NAME:
The pathname of a device or file where the job command stream resides.

LISTING ACCESS NAME:
The pathname of a device or file where the job execution results are to be listed.

*Example:*

In the following example, suppose a batch job is created that creates a file, outputs data to the file, prints the file contents, then deletes the file. The command stream to perform these functions resides in a file named SYS1.KC0017.INPUT, and the logical names and synonyms of the creating job will be passed to the new job. The XBJ command could be used to create and execute the batch job as shown below:

```
[ ] XBJ
EXECUTE BATCH SCI JOB
                    JOB NAME:  BATCH
          USE CURRENT USER ID?:  YES
LOGICAL NAME TABLE PATHNAME:
     SYNONYM TABLE PATHNAME:
           INPUT ACCESS NAME:  SYS1.KC0017.INPUT
         LISTING ACCESS NAME:  LP01
```

### 5.3.4 Operating from Card Reader

To execute a batch stream on a deck of cards in the card reader, the macro assembly stream should be in the prescribed order as shown in Figure 5-9.

Execute the assembly stream by entering the XBJ command and entering CR01 in response to the INPUT ACCESS NAME prompt, as shown below:

```
[ ] XBJ

EXECUTE BATCH JOB
                    JOB NAME:  BATCH
          USE CURRENT USER ID?:  YES
LOGICAL NAME TABLE PATHNAME:
   SYNONYM TABLE PATHNAME:
          INPUT ACCESS NAME:  CR01
       LISTING ACCESS NAME:  .USER.LFILE
```

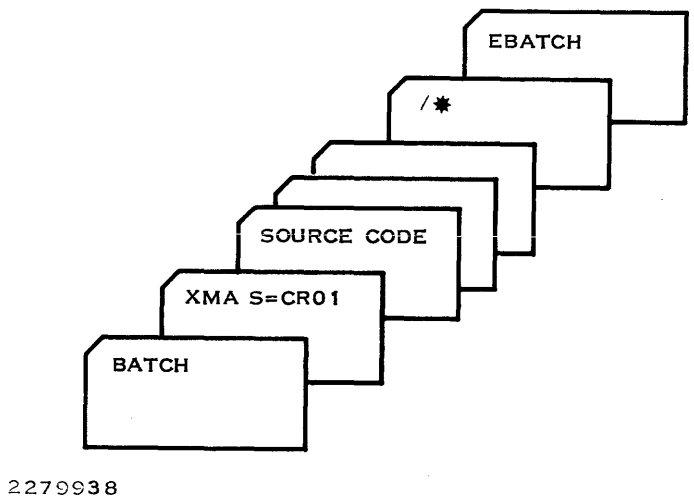



**Figure 5-9. Macro Assembly Stream for Cards**

# 6

# Linking and Installing a Program

## 6.1 SUPPORTED FEATURES

The Link Editor links separate object modules together to form a single program which runs under DNOS.

The following Link Editor features are supported by DNOS:

- Automatic overlay loading
- Random libraries
- Sequential libraries
- ASCII, compressed and image format
- Absolute memory partitioning.

For more information about these features, consult the *DNOS Link Editor Reference Manual.*

## 6.2 LINK EDIT CONTROL FILE

The first step in performing a link edit run is to write a control file defining the link edit functions. The control file can be written using the DNOS Text Editor. The control file contains link edit commands and the names of any object modules. Object modules not included in the control file may be on disk, tape, cassette, cards, or diskette.

Table 6-1 presents a brief description and syntax for the Link Editor commands. Refer to the *Link Editor Reference Manual* for complete details on the commands.

**Table 6-1. Link Editor Commands**

| Command | Description |
|---|---|
| | **Symbol Resolution Commands** |
| LIBRARY | Specifies the libraries to be searched for unresolved external references |
| AUTO | Specifies use of automatic symbol resolution |
| NOAUTO | Inhibits use of automatic symbol resolution |
| SEARCH | Specifies that the symbols in the random or sequential libraries specified are to be resolved when this command is issued |
| FIND | Specifies that the symbols in the random or sequential libraries specified are to be resolved when this command is issued |
| | **Procedure, Task and Overlay Linking Commands** |
| PROCEDURE | Defines a phase be installed as a procedure |
| TASK | Defines a phase be installed as a task |
| PHASE | Specifies a new object module in the linked object file and states the level and name of the phase |
| ALLOCATE | Controls the relative position of program, data, and common segments |
| LOAD | Includes the overlay manager when the FORMAT IMAGE command is used |
| NOLOAD | Specifies that the overlay manager and its tables be excluded from the linked output |
| SHARE | Specifies modules to share the same data area |
| PARTIAL | Performs a partial link edit and outputs a normal tagged object or compressed tagged object output file |
| NOTGLOBAL | Identifies symbols defined in the current phase as not global |
| ALLGLOBAL | Declares all external definitions in the current phase to be global symbols |
| GLOBAL | Identifies symbols defined in the current phase as global |
| DUMMY | Suppresses the linked output for the defined phase, procedure, or task in which it appears |

**Table 6-1. Link Editor Commands (Continued)**

| Command | Description |
|---|---|
| **Procedure, Task and Overlay Linking Commands (Continued)** | |
| ADJUST | Specifies alignment of a phase or module within a phase |
| | **Symbol Processing Commands** |
| SYMT | Includes the symbol tables in the linked object module |
| NOSYMT | Omits the symbol tables from the linked object module |
| | **Execution and Listing Option Commands** |
| INCLUDE | Defines the modules or files of modules to be included in the linked object output |
| FORMAT | Defines the format of the linked output (normal tagged object, compressed tagged object, or memory image format) |
| MAP | Controls the format of the link map |
| NOMAP | Suppresses the load map listing |
| PAGE/NOPAGE | Controls the format of the output listing |
| ERROR/NOERROR | Specifies whether link editor is to continue after it encounters an error |
| | **Absolute Memory Partitioning Commands** |
| PROGRAM | Specifies the starting location counter value for the program segments |
| DATA | Defines the starting location counter value for data segments |
| COMMON | Defines the starting location counter value for the specified common segments |

## 6.3 LINK EDITOR OPERATION WITH DNOS

The Link Editor is executed by entering the Execute Link Editor (XLE) command. Enter XLE and press the RETURN key. The following appears:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:  pathname@       (*)
 LINKED OUTPUT ACCESS NAME:   [pathname@]     (*)
         LISTING ACCESS NAME: [pathname@]     (*)
          PRINT WIDTH (CHARS): integer        (80)
```

After entering the last response to the prompts, enter the WAIT command. The message

```
I LINKER-0001 LINK EDITOR COMPLETED, 0 ERRORS, 0 WARNINGS:
```

appears on the screen when the linking process terminates. Press the CMD key to return to the command mode.

The prompts for the XLE command are described below:

CONTROL ACCESS NAME:
: The pathname of the Link Editor control file. The control file can be on a sequential disk file, or any sequential device such as a tape unit, cassette unit, or card reader.

LINKED OUTPUT ACCESS NAME:
: The access name of the sequential device or file where the output of the Link Editor is written. If the object output is not desired, the user may specify DUMY which will suppress the generation of the output. Use of the DUMY value allows for a trial run to ensure that no errors occur.

  If the FORMAT command specifies the IMAGE option, the entry made in response to the LINKED OUTPUT ACCESS NAME prompt must be a DNOS program file or a DNOS image file.

LISTING ACCESS NAME:
: The access name of the device or file where the load map listing is written. If the listing output is not desired, the user may specify DUMY which will suppress the listing. The value entered in response to the prompt can be any valid DNOS access name, synonym, or device name.

  For a description of the load map listing, refer to the *DNOS Link Editor Reference Manual.*

PRINT WIDTH (CHARS):
: The width of the print line.

The following example shows the responses for the prompts when the control file is on .USER.CNTRLINK, the output file is .USER.LNKOUT, the listing device is line printer one (LP01), and the initial PRINT WIDTH (CHARS): value is accepted:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:   .USER.CNTRLINK
 LINKED OUTPUT ACCESS NAME:   .USER.LNKOUT
          LISTING ACCESS NAME:   LP01
            PRINT WIDTH (CHARS):   80
```

## 6.4 PROGRAM LINKING AND DIRECTIVES

Since the assembler includes directives that generate the information required to link program modules, it is not necessary to assemble an entire program in the same assembly. A long program may be divided into separately assembled modules to avoid a long assembly or to reduce the symbol table size. Also, modules common to several programs may be combined as required. Program modules may be linked by the Link Editor to form a linked object module that may be stored on a library and/or loaded as required. The following paragraphs define the linking information that must be included in a program module.

### 6.4.1 External Reference Directives

Each symbol from another program module must be placed in the operand field of an REF or SREF directive in the program module that requires the symbol.

### 6.4.2 External Definition Directive

Each symbol defined in a program module and required by one or more other program modules must be placed in the operand field of a DEF directive.

### 6.4.3 Program Identifier Directive

Program modules linked by the Link Editor should include an IDT directive. The module name specified in the IDT directive should be unique.

### 6.4.4 Linking Program Modules

The Link Editor matches symbols from REF directives and symbols from DEF directives in a similar manner within a program phase. The Link Editor follows linking commands to determine the modules to be linked. IDT character strings are not matched with REF directive operands.

## 6.5 LINK MAP

Figure 6-1 shows the DNOS format of the output listing generated by the Link Editor. This example linked three modules to form a task. The three modules are named SUBT1, SUBR1, and MODX, and the task itself is named LSCAN.

```
LINKER                                                          PAGE  1
 COMMAND LIST

 TASK   LSCAN
 LIBRARY      .MSK.EXO
 INCL   .SUBT1
 INCL   .SUBR1
 INCL   .MODX
 END

 LINKER                                                         PAGE  2
 LINK MAP

 CONTROL FILE = .MSK.EXO.MODCOM

 LINKED OUTPUT FILE = DUMY

 LIST FILE = .MSK.LST

 OUTPUT FORMAT = ASCII


 LINKER                                                         PAGE  3

 PHASE 0, LSCAN    ORIGIN = 0000  LENGTH = 0056    ENTRY = 0000


 MODULE   NO  ORIGIN   LENGTH     TYPE      DATE       TIME      CREATOR

 SUBT1     1    0000     0034    INCLUDE  04/26/80  13:27:49  SDSMAC
 SUBR1     2    0034     000C    INCLUDE  04/26/80  13:30:29  SDSMAC
 MODX      3    0040     0016    INCLUDE  04/26/80  13:33:35  SDSMAC

                        D E F I N I T I O N S

 NAME   VALUE NO   NAME   VALUE NO   NAME  VALUE NO   NAME  VALUE NO
 DC$AMP 002A  1    DC$RET 002C  1    DC$TX 002E  1    *MODX 0040  3
 *SUBR1 0034  2    *SUBT1 0000  1

              U N R E S O L V E D   R E F E R E N C E S

 NAME   COUNT NO   NAME  COUNT NO   NAME  COUNT NO   NAME  COUNT NO

 SUBR   1      1


 ****  LINKING COMPLETED
```

**Figure 6-1. Link Edit Output Listing**

Page one in the example, titled COMMAND LIST, is the list of the control file used to control the linking operations. This list is generated at the beginning of each Link Edit. Page two, titled LINK MAP, lists the parameters entered at the terminal when the Link Editor was activated. This page also gives the format of the output from the Link Editor (ASCII in the example). The last page, page three, is the actual link map. The PHASE name, address of the ORIGIN, LENGTH of the linked object code, and the execution ENTRY point are defined in the top line.

The subdivisions of the link map are listed below:

MODULE
The module names (identified by the IDT directives) included in the phase.

ORIGIN
The beginning of the module relative to the beginning of the program.

LENGTH
The length of the module, in bytes.

TYPE
The method by which the module was included in the phase (INCLUDE, SEARCH command, LIBRARY auto resolution).

TIME
The time the module was created.

CREATOR
The assembler or compiler that generated the module (SDSMAC).

DEFINITIONS
The entries under this heading describe all external definitions (DEFs) in the phase.

NAME — The symbol specified by the DEF statement.

VALUE — The address within the phase associated with the symbol.

NO — The number of the module within the phase in which the symbol is DEFed.

**NOTE**

Names that are DEFed within the phase but not referenced (REFed) within the program are preceded by an asterisk (*). Symbols that are self-defining (absolute) are identified by a trailing asterisk (*).

UNRESOLVED REFERENCES

The entries under this section of the listing defines any references that are unresolved within the phase.

NAME — The symbol that was referenced and could not be found.

COUNT — The number of times the symbol was referenced.

NO — The module within the phase in which the reference occurred.

Unresolved references cause a warning message to be output at the end of the link map. The message is of the form:

n WARNINGS

where n is the number of unresolved references.

**NOTE**

Partial link edits do not produce a warning message for unresolved references.

The end of the Link Edit processing is indicated by the following message:

**** LINKING COMPLETED

## 6.6 LINK EDITOR EXAMPLES

The following paragraphs contain examples of Link Edits on a DNOS system. Provided for each example is the complete Link Map (containing a copy of the control file) and the parameters entered when the Link Editor is called from a VDT.

### 6.6.1 Single Task With No Procedure — Example

The example shown in Figure 6-2 illustrates the use of the Link Editor to build a task consisting of two modules with no attached procedures. The parameters entered in response to the prompts are as follows:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:   .USER.TEST1
 LINKED OUTPUT ACCESS NAME:   DUMY
          LISTING ACCESS NAME:   .USER.TEST1L
            PRINT WIDTH (CHARS):  80
```

Note that no linked output is created since the LINKED OUTPUT ACCESS NAME: DUMY, was used. The default value was used in response to the PRINT WIDTH (CHARS) prompt.

```
LINKER                                                          PAGE  1
COMMAND LIST

TASK    RANDOM
INCLUDE       .USER.TESTX
INCLUDE       .USER.SORT
END


LINKER                                                          PAGE  2
LINK MAP

CONTROL FILE = .USER.TEST1

LINKED OUTPUT FILE = DUMY

LIST FILE = .USER.TEST1L

OUTPUT FORMAT = ASCII


LINKER                                                          PAGE  3

PHASE 0, RANDOM    ORIGIN = 0000    LENGTH = 005E    ENTRY = 0000


MODULE    NO  ORIGIN  LENGTH    TYPE       DATE      TIME      CREATOR

TESTX      1    0000    0032    INCLUDE   04/26/80  13:09:29  SDSMAC
SORT       2    0032    002C    INCLUDE   04/26/80  13:12:48  SDSMAC

                              DEFINITION

NAME  VALUE NO    NAME  VALUE NO    NAME  VALUE NO    NAME    VALUE NO

SORT  0032  2

****  LINKING COMPLETED
```

**Figure 6-2. Single Task, No Procedure Example**

The control file defines the task name as being RANDOM, with files TESTX and SORT included by use of the INCLUDE command. The default format, ASCII, is used.

The Link Map shows that PHASE 0, RANDOM, begins at relative address 0000 and has a length of >005E bytes. Module TESTX is >32 bytes in length and begins at relative address 0000, and module SORT is >2C bytes in length and begins at relative address >32.

Only one external definition, SORT, is made.

### 6.6.2 Task with Two Attached Procedures — Example

The example shown in Figure 6-3 is a Link Edit for a program having a task, CONTRL, and two attached procedures, TABLE and ROUT. The parameters entered when the Link Editor is activated from the VDT are as follows:

```
[ ] XLE

EXECUTE LINK EDITOR
        CONTROL ACCESS NAME:   .USER.EXC.TWOP
 LINKED OUTPUT ACCESS NAME:   DUMY
          LISTING ACCESS NAME:   .USER.LST
            PRINT WIDTH (CHARS):  80
```

Note that within the control file, the procedures are defined before the task. On the Link Map, the procedures are also presented first. Page three of the example contains the Link Map for Procedure 1, TABLE, which has an origin at relative address 0000 and a length of eight bytes. One module, ALPHA, is included in TABLE and it is taken from random library .USER.EXO.

Procedure 2, ROUT, is shown in the Link Map on page four of the example and consists of one module, BETA, which has a relative origin of >20 and a length of eight bytes. BETA is specified by an INCLUDE command and is read from the random library .USER.EXO. Note that BETA contains one external definition, B$BY, that is not referenced. External definitions that are not referenced are denoted by a preceding asterisk (*).

PHASE 0, shown on page five of the example, is defined by the TASK command and is named CONTRL. CONTRL consists of one module, TGAMA, specified by the INCLUDE command and read from the random library .USER.EXO. CONTRL has an origin at relative address >40 and a length of >3C bytes. CONTRL contains no external definitions.

The two procedures have to be installed before the task is installed using the Install Procedure (IP) and the Install Task (IT) commands, respectively.

The output format of the Link Edit is ASCII.

```
LINKER                                                          PAGE 1
COMMAND LIST

LIBRARY       .USER.EXO
PROCEDURE     TABLE
INCL    (ALPHA)
PROC    ROUT
INCLUDE (BETA)
TASK    CONTRL
INCL    (TGAMA)
END



LINKER                                                          PAGE  2
LINK MAP

CONTROL FILE = .USER.EXC.TWOP

LINKED OUTPUT FILE = DUMY

LIST FILE = .USER.LST

OUTPUT FORMAT = ASCII

LIBRARIES

NO    ORGANIZATION  PATHNAME

1     RANDOM        .USER.EXO

LINKER                                                          PAGE  3

PROCEDURE 1,  TABLE     ORIGIN = 0000    LENGTH = 0008

MODULE    NO  ORIGIN   LENGTH   TYPE       DATE       TIME     CREATOR

ALPHA     1   0000       0008  INCLUDE, 1 04/26/80  13:52:07   SDSMAC


                          DEFINITIONS

NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE NO

M$A   0000  1    M$B   0002  1    M$C   0004  1    M$D   0006  1
```

**Figure 6-3. Task, Two Attached Procedures Example (Sheet 1 of 2)**

```
LINKER                                                          PAG

PHASE 0, T$CAL  ORIGIN = 0040   LENGTH = 00A0   ENTRY = 0000

MODULE   NO  ORIGIN  LENGTH     TYPE      DATE      TIME      CREA
ROOT     2   0040    0050    INCLUDE   04/26/80  15:20:37  SDSM

LINKER                                                          PAG

PHASE 1, O$ONEA  ORIGIN = 0C90   LENGTH = 0028  ENTRY = 0000

MODULE   NO  ORIGIN  LENGTH     TYPE      DATE      TIME      CREA
MOD1     3   0090    0028    INCLUDE   04/26/80  15:20:37  SDSM

                          DEFINITIONS

NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE
SUBR1 0090  3

LINKER                                                          PAG

PHASE 2, O$TWOA  ORIGIN = 00B8   LENGTH = 0028  ENTRY = 0000

MODULE   NO  ORIGIN  LENGTH     TYPE      DATE      TIME      CREA
MOD2     4   00B8    0028    INCLUDE   04/26/80  15:31:12  SDSM

                          DEFINITIONS

NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE NO   NAME  VALUE
SUBR2 00B8  4
```

Figure 6-4. Overlaid Program Example (Sheet 2 of 3)

```
LINKER                                                          PAGE  7

PHASE 2,  O$TWOB   ORIGIN = 00B8   LENGTH = 0028   ENTRY = 0000

MODULE    NO   ORIGIN   LENGTH      TYPE      DATE      TIME      CREATOR

MOD3      5     00B8     0028    INCLUDE    04/26/80 15:31:50    SDSMAC


                              DEFINITIONS

NAME   VALUE NO    NAME   VALUE NO    NAME   VALUE NO    NAME   VALUE NO

SUBR3 00B8  5
```

```
LINKER                                                          PAGE  8

PHASE 1,  O$ONEB    ORIGIN = 0090   LENGTH = 0034


MODULE    NO   ORIGIN   LENGTH      TYPE      DATE      TIME      CREATOR

MOD4      6     0090     002C    INCLUDE   04/26/80  15:40:57   SDSMAC
MODDAT    7     00BC     0008    INCLUDE   04/26/80  15:47:16   SDSMAC


                              DEFINITIONS

NAME   VALUE NO    NAME   VALUE NO    NAME   VALUE NO    NAME   VALUE NO

SUBR4 0090  6     TABLE 00BC  7


****  LINKING COMPLETED
```

**Figure 6-4. Overlaid Program Example (Sheet 3 of 3)**

## 6.7 LINKED FORMAT OUTPUT OPTIONS

The following paragraphs define the listing output options. The link edit FORMAT command defines the format of the linked object code.

The syntax of the FORMAT command is as follows:

```
                ASCII
FORMAT          COMPRESSED
                IMAGE          [,REPLACE]      [,priority]
```

There are three formats supported by the Link Editor: normal tagged object, compressed tagged object, and memory image format. The default is 4.

### 6.7.1 Normal Tagged Object

This format consists of ASCII characters and ASCII control tag characters. Except for COBOL, it must be output to a sequential file. Except for COBOL, the normal tagged object is not executable and must be installed or loaded as a task/procedure/overlay before it can be executed. Normal tagged object format is generally transportable between 990 computer systems and can be linked again if generated using a PARTIAL command. Normal tagged object is the default value for the FORMAT command.

### 6.7.2 Compressed Tagged Object

This format is a condensed version of the normal tagged object and can only be output to a file that supports binary data. Except for this, compressed object is treated as normal tagged object. Compressed object results in a savings of disk space as compared to the normal tagged format. The difference between compressed and normal object is that in compressed the numeric fields are expressed in binary instead of ASCII. Also, in compressed format, the binary 01 is used instead of tag 0.

### 6.7.3 Memory Image Format

Memory image format appears exactly as the program appears in memory and is loaded directly to a DNOS Program File or a DNOS Image File.

When the IMAGE format is selected, the user may enter the REPLACE parameter which causes the new procedures, tasks, or overlays to replace existing ones of the same name, in the program file (defined by the LINKED OUTPUT ACCESS NAME). The task execution priority is defined by the priority parameters (1,2,3 or 4). The default priority value is 4.

In DNOS, the IMAGE format can also be used to install the Linked Output on an Image File, a unique file type containing the loadable image for the loader. It is used for the Initial Program Load. Refer to following paragraphs on installing the image file with the Link Editor.

The Link Editor cannot be used to install memory-resident, system, or privileged tasks on a program file. These tasks must be installed using the Install Task SVC, or the Install Task (IT) command.

## 6.8 INSTALLING A LINKED PROGRAM

Under DNOS, programs are called tasks. A task may be segmented to include sharable procedures and may also include overlays. After link edit, and before program execution, the task and its procedures and overlays must be installed on a program file (unless this step is bypassed by use of the IMAGE format option of the Link Editor). Either the system program file or a program file created by the Create Program File (CFPRO) command may be used to install the task. To create the program file, .USER.PROGA for the example program, enter CFPRO and press RETURN. Enter the responses displayed below:

```
[ ] CFPRO

CREATE PROGRAM FILE
                    PATHNAME:  .USER.PROGA
          MAX NUMBER OF TASKS:  255
MAX NUMBER OF PROCEDURES:  255
     MAX NUMBER OF OVERLAYS:  255
           INITIAL ALLOCATION:  85
       SECONDARY ALLOCATION:
                 EXPANDABLE?:  YES
```

All of the install commands in this section allow the program file and the object file to be specified by file name or by LUNO. The manner in which the program file is selected is arbitrary. There is an important difference between selecting the object file by LUNO and selecting the object file by pathname. Files specified by pathname are rewound when opened, but files specified by LUNO are not rewound when opened. Therefore, if the same object file contains procedures, tasks, and overlays, it must be specified by LUNO for the commands to correctly install all of the object code in a program.

Procedures, tasks and overlays must be installed in the following order:

1. Procedures, if any, must be installed first.
2. The task is installed after the procedures.
3. Overlays are installed last.

Thus, object files containing more than one object (procedure, task, overlay) must be ordered with the procedures first, task second, and overlays last.

To install object files, enter the installation commands necessary and answer the prompts (pressing the RETURN key after each response).

Do not install a task on the S$UTIL file. Use the program file .S$SHARED to install shared procedures. The .S$SHARED program file is used as the default program file for a bid with no LUNO specified. It is recommended that you install tasks in your own program file. This recommendation also applies to installing real-time tasks, procedures, and overlays.

The following paragraphs discuss the commands which install, delete, and modify programs. For complete command descriptions, refer to the *DNOS System Command Interpreter (SCI) Reference Manual.*

Installing or modifying a task or procedure to be memory resident requires an Initial Program Load (IPL) be performed before the task or procedure is in memory.

### 6.8.1 Install Task Segment — IT

The IT command places an executable task on a program file. If the task has attached procedures, the procedures must be installed before the IT command. For an explanation of the task attributes (priority, privileged, system, memory resident, and replicatable) consult Section 3.

The IT command and prompts are described below:

*Prompts:*

| | | |
|---|---|---|
| INSTALL TASK SEGMENT | | |
| PROGRAM FILE OR LUNO: | {filename@/integer} | (*) |
| TASK NAME: | [alphanumeric] | |
| TASK ID: | [integer] | |
| OBJECT PATHNAME OR LUNO: | {pathname@/integer} | (*) |
| PRIORITY: | [integer] | (4) |
| DEFAULT TASK FLAGS?: | YES/NO | (YES) |
| ATTACHED PROCEDURES?: | YES/NO | (NO) |

If the response to the DEFAULT TASKS FLAGS?: prompt was NO, the following sets of prompts are displayed on the user's terminal:

| | | |
|---|---|---|
| DEFINE TASK FLAGS | | |
| PRIVILEGED?: | YES/NO | (NO) |
| SYSTEM TASK?: | YES/NO | (NO) |
| MEMORY RESIDENT?: | YES/NO | (NO) |
| REPLICATABLE?: | YES/NO | (YES) |
| DELETE PROTECT?: | YES/NO | (NO) |
| IN MEMORY COPYABLE?: | YES/NO | (NO) |
| IN MEMORY REUSABLE?: | YES/NO | (NO) |
| UPDATABLE?: | YES/NO | (NO) |
| SOFTWARE PRIVILEGED?: | YES/NO | (NO) |

After the responses to the previous prompts are entered, the following prompts are displayed on the user's terminal:

| | | |
|---|---|---|
| 990/12 FLAGS | | |
| EXECUTE PROTECT?: | YES/NO | (NO) |
| OVERFLOW CHECKING?: | YES/NO | (NO) |
| WRITABLE CONTROL STORAGE?: | YES/NO | (NO) |

If the response to the ATTACHED PROCEDURES: prompt was YES, the following set of prompts is displayed on the user's terminal:

| | | |
|---|---|---|
| ATTACH TASK PROCEDURES | | |
| 1ST PROCEDURE ID: | integer | (0) |
| P1 FROM TASKS PROGRAM FILE?: | [YES/NO] | (YES) |
| 2ND PROCEDURE ID: | [integer] | (0) |
| P2 FROM TASKS PROGRAM FILE?: | [YES/NO] | (YES) |

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name of or the LUNO assigned to the program file on which the task segment is to be installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to the execution of the IT command. If zero is specified, the .S$SHARED program file is assumed.

TASK NAME:

A user-defined character string, up to eight ASCII characters, which is the name of the task segment to be installed on the specified program file. If zero or a null response is specified, the system will assign the IDT name of the object module as the task segment name.

TASK ID:

A hexadecimal value which will be associated with the installed task segment. If zero or a null response is specified, the system assigns an available ID.

OBJECT PATHNAME OR LUNO:

The pathname of or the LUNO assigned to a device or file where the object module of the task segment resides.

PRIORITY:

The integer value which represents the execution priority level of the task. Priorities may range from zero through four. Priorities zero through three are fixed, with priority zero as the highest level and three the lowest. Priority four is dynamically managed by the operating system. Four is the default priority level.

DEFAULT TASK FLAGS?:

If NO is entered in response to this prompt, the user has the option to set the task flags. If YES is entered, the initial values are used for the flags.

PRIVILEGED?:

If YES is entered, the task is allowed to execute privileged hardware instructions. Privileged hardware instructions should be executed cautiously and only by the user who is very familiar with the system.

SYSTEM TASK?:

If YES is entered, the task is allowed to execute in system memory space. Tasks should be executed in system memory space with caution and only by the user who is very familiar with the system.

MEMORY RESIDENT?:

If YES is entered, and the task is installed on the program file .S$SHARED of the applications program file specified at system generation the task will be loaded into memory during initial program load (IPL) and remain in memory when terminated.

REPLICATABLE?:

If YES is entered, there may be multiple copies of the task in memory simultaneously.

DELETE PROTECT?:
If YES is entered, the task segment cannot be deleted from the program file unless the Modify Task Segment Entry (MTE) command is used to unprotect the task segment prior to the execution of the Delete Task Segment (DT) command. If NO is specified, the task segment may be deleted by the DT command.

IN MEMORY COPYABLE?:
If YES is entered, the task segment may be copied from memory rather than being copied from disk. This situation may occur if the task is in memory, and another user wishes to execute the task.

IN MEMORY REUSABLE?:
If YES is entered, the task segment memory may be reused by another task rather than being copied from disk or from one memory location to another.

UPDATABLE?:
If YES is entered, the data of a task may be modified, and the task segment will be written to disk with the new data modifications when the task terminates.

SOFTWARE PRIVILEGED?:
If YES is entered, the task is allowed to execute privileged supervisor calls.

EXECUTE PROTECT?:
If YES is entered, execution of the task segment is prohibited. The protection is enforced only on a 990/12 computer.

OVERFLOW CHECKING?:
If YES is entered, the occurrence of arithmetic overflow will cause control of the task to pass to the task's end action routine. Overflow checking is enforced only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the task uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

ATTACHED PROCEDURES?:
If YES is entered in response to this prompt, the user will be prompted for the ID(s) of procedure segments attached to this task segment and asked if the procedures reside on the same program file as the task.

1ST PROCEDURE ID:
The integer value representing the ID of a procedure attached to the task segment. If zero is entered, there are no procedures.

P1 FROM TASKS PROGRAM FILE?:
If YES is entered, the attached procedure segment whose ID was specified for the 1ST PROCEDURE ID: prompt resides on the same program file as the task segment. If NO is entered, that procedure segment must reside on the .S$SHARED program file.

2ND PROCEDURE ID:
The integer value representing the ID of a procedure segment attached to the task segment. If zero is entered, there is no 2nd procedure.

P2 FROM TASKS PROGRAM FILE?:

If YES is entered, the attached procedure segment whose ID was specified for the 2ND PROCEDURE ID: prompt resides on the same program file as the task. If NO is entered, that procedure segment must reside on the .S$SHARED program file.

### 6.8.2 Install Real-Time Task Segment — IRT

The IRT command places an executable real-time task on a program file. If the task has attached procedures, the procedures must be installed before the IRT command. For an explanation of the task attributes (priority, privileged, system, memory resident, and replicative) consult Section 3. The previously mentioned restrictions on installing tasks also apply to installing real-time tasks. The prompts for installing a real-time task are similar to the IT command and are described in the paragraph on installing a task.

*Prompts:*

```
INSTALL REAL-TIME TASK SEGMENT
      PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
                TASK NAME:   [alphanumeric]
                  TASK ID:   [integer]
  OBJECT PATHNAME OR LUNO:   {pathname@/integer}      (*)
                PRIORITY:    integer
       DEFAULT TASK FLAGS?:  YES/NO                   (YES)
     ATTACHED PROCEDURES?:   YES/NO                   (NO)
```

If the response to the DEFAULT TASKS FLAGS?: prompt was NO, the following prompts are displayed on the user's terminal:

```
DEFINE TASK FLAGS
          PRIVILEGED?:  YES/NO   (NO)
          SYSTEM TASK?: YES/NO   (NO)
       MEMORY RESIDENT?: YES/NO  (NO)
         REPLICATABLE?: YES/NO   (YES)
       DELETE PROTECT?: YES/NO   (NO)
  IN MEMORY COPYABLE?:  YES/NO   (NO)
  IN MEMORY REUSABLE?:  YES/NO   (NO)
          UPDATABLE?:   YES/NO   (NO)
  SOFTWARE PRIVILEGED?: YES/NO   (NO)
```

After the responses are entered for the previous prompts, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
        EXECUTE PROTECT?:  YES/NO   (NO)
      OVERFLOW CHECKING?:  YES/NO   (NO)
WRITABLE CONTROL STORAGE?: YES/NO   (NO)
```

If the response to the ATTACHED PROCEDURES: prompt was YES, the following prompts are displayed on the user's terminal:

```
ATTACH TASK PROCEDURES
            1ST PROCEDURE ID:  integer          (0)
P1 FROM TASKS PROGRAM FILE?:   [YES/NO]         (YES)
            2ND PROCEDURE ID:  [integer]        (0)
P2 FROM TASKS PROGRAM FILE?:   [YES/NO]         (YES)
```

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name or the LUNO assigned to the program file on which the task segment is to be installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the IRT command. If zero is specified, the .S$SHARED program file is assumed.

TASK NAME:

A user-defined character string, up to eight ACSII characters, which is the name of the task segment to be installed on the specified program file. If zero or a null response is entered, the system assigns the IDT name of the object module as the name of the task segment.

TASK ID:

A hexadecimal value which will be associated with the installed task segment. If the response to this prompt is zero or a null response is entered, the system assigns an available ID.

OBJECT PATHNAME OR LUNO:

The pathname of or the LUNO assigned to the device or file where the object module for the task segment resides.

PRIORITY:

The integer value which represents the execution priority level of the task segment. Priorities may range from 1 through 127, with 1 being the highest priority.

DEFAULT TASK FLAGS?:

If NO is entered in response to this prompt, the user has the option to set the task flags. If YES is entered, the initial values are used for the flags.

PRIVILEGED?:

If YES is entered, the task is allowed to execute privileged hardware instructions. Privileged hardware instructions should be executed cautiously and only by the user who is very familiar with the system.

SYSTEM TASK?:

If YES is entered, the task is allowed to execute in system memory space. Tasks should be executed in system memory space with caution and only by the user who is very familiar with the system.

MEMORY RESIDENT?:
If YES is entered, and the task is installed on the program file .S$SHARED or the application program file specified at system generation, the task will be loaded into memory during initial program load (IPL) and remain in memory when terminated.

REPLICATABLE?:
If YES is entered, there may be multiple copies of the task in memory simultaneously.

DELETE PROTECT?:
If YES is entered, the task segment cannot be deleted from the program file unless the Modify Task Segment Entry (MTE) command is used to unprotect the task segment prior to the execution of the Delete Task Segment (DT) command. If NO is specified, the task segment may be deleted by executing the DT command.

IN MEMORY COPYABLE?:
If YES is entered, the task segment may be copied from memory rather than being copied from disk. This situation may occur if the task is in memory and another user wishes to execute the task.

IN MEMORY REUSABLE?:
If YES is entered, the task segment memory may be reused by another task rather than being copied from disk or from one memory location to another.

UPDATABLE?:
If YES is entered, the data of the task segment may be modified, and the task segment will be written to disk with the new data modifications after the task terminates.

SOFTWARE PRIVILEGED?:
If YES is entered, the task is allowed to execute privileged supervisor calls. Privileged supervisor calls should be executed cautiously and only by the user who is very familiar with the system.

EXECUTE PROTECT?:
If YES is entered, execution of the task is prohibited. The protection is enforced only on a 990/12 computer.

OVERFLOW CHECKING?:
If YES is entered, the occurrence of arithmetic overflow causes control of the task to pass to the task's end action routine. Overflow checking is available only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the task uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

ATTACHED PROCEDURES?:
If YES is entered in response to this prompt, the user will be prompted for the ID(s) of procedure segments attached to this task segment and asked if the procedures reside on the same program file as the task.

1ST PROCEDURE ID:
The integer value representing the ID of a procedure segment attached to the task segment. If zero is entered, there are no procedures.

P1 FROM TASKS PROGRAM FILE?:
If YES is entered, the attached procedure segment whose ID was specified for the 1ST PROCEDURE ID: prompt resides on the same program file as the task segment. If NO is entered, that procedure segment must reside on the .S$SHARED program file.

2ND PROCEDURE ID:
The integer value representing the ID of a procedure segment attached to the task segment. If zero is entered, there is no 2nd procedure segment.

P2 FROM TASKS PROGRAM FILE?:
If YES is entered, the attached procedure segment whose ID was specified for the 2ND PROCEDURE ID: prompt resides on the same program file as the task segment. If NO is entered, that procedure segment must reside on the .S$SHARED program file.

### 6.8.3 Install Procedure Segment — IP

The IP command places a procedure on a program file and assigns a procedure ID for use by subsequent IT commands. The previously mentioned restrictions on installing tasks also apply to installing procedures. The IP command and prompts are described below:

*Prompts:*

```
INSTALL PROCEDURE SEGMENT
        PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
              PROCEDURE NAME:  [alphanumeric]
                PROCEDURE ID:  [integer]
  OBJECT PATHNAME OR LUNO:     {pathname@/integer}      (*)
          MEMORY RESIDENT?:    YES/NO                   (NO)
          DELETE PROTECT?:     YES/NO                   (NO)
```

After responses are entered for the previous prompts, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
          EXECUTE PROTECT?:    YES/NO                   (NO)
            WRITE PROTECT?:    YES/NO                   (NO)
WRITABLE CONTROL STORAGE?:     YES/NO                   (NO)
```

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of, or the LUNO assigned to, the program file on which the procedure segment is to be installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the IP command. If zero is specified, the .S$SHARED program file is assumed.

PROCEDURE NAME:
A user-defined character string, up to eight characters, that identifies the procedure segment. If the procedure name is not specified, the system will assign the IDT name of the object module as the procedure name.

PROCEDURE ID:
A hexadecimal integer that will be assigned as the ID of the procedure segment. If zero or a null response is specified, the system assigns an ID.

OBJECT PATHNAME OR LUNO:
The name of, or the LUNO assigned to, a device or file where the object module for the procedure segment resides.

MEMORY RESIDENT?:
If YES is entered, and the procedure segment is installed on the program file .S$SHARED or the applications program file specified at system generation, the procedure segment will be loaded into memory during initial program load (IPL) and will stay in memory even when terminated.

DELETE PROTECT?:
If YES is entered, the procedure segment cannot be deleted from the program file unless the Modify Procedure Segment Entry (MPE) command is used to unprotect the procedure segment prior to the execution of the Delete Procedure Segment (DP) command. If NO is specified, the procedure segment may be deleted by executing the DP command.

EXECUTE PROTECT?:
If YES is entered, the procedure segment cannot be executed. The protection is enforced only on a 990/12 computer.

WRITE PROTECT?:
If YES is entered, the procedure segment cannot be modified when in memory. The protection is enforced only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the procedure segment uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

### 6.8.4 Install Overlay — IO

The IO command places an overlay associated with a task on the program file with the task. The task must be installed before the overlay and may be specified by name or installed ID. The previously mentioned restrictions on installing tasks also apply to installing real-time tasks. The IO command and prompts are described below:

*Prompts:*

```
INSTALL OVERLAY
        PROGRAM FILE OR LUNO:  {filename@/integer}          (*)
                OVERLAY NAME:  [alphanumeric]
                  OVERLAY ID:  [integer]
  OBJECT PATHNAME OR LUNO:     {pathname@/integer}          (*)
                RELOCATABLE?:  YES/NO                       (NO)
              DELETE PROTECT?: YES/NO                       (NO)
ASSOCIATED TASK NAME OR ID:    [{character(s)/integer}]     (*)
```

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of, or LUNO assigned to, the program file on which the overlay is to be installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to the execution of the IO command. If zero is specified, the .S$SHARED program file is assumed.

OVERLAY NAME:
A user-defined character string, a maximum of eight characters, that is unique to the program file. If a null response is specified, the system uses the IDT name of the object module as the name of the overlay.

OVERLAY ID:
An integer value in the range of 1 through 255 that is associated with the overlay name and is unique to the program file. If zero or a null response is specified, the system will assign an ID to the overlay.

OBJECT PATHNAME OR LUNO:
The name of, or the LUNO assigned to, the device or file where the object module for the overlay resides.

RELOCATABLE?:
If YES is entered, the overlay is allowed to be loaded at an address other than its natural load address.

DELETE PROTECT?:
If YES is entered, the overlay cannot be deleted from the program file unless the Modify Overlay Entry (MOE) command is used to unprotect the overlay prior to the execution of the Delete Overlay (DO) command. If NO is specified, the overlay may be deleted by executing the DO command.

ASSOCIATED TASK NAME OR ID:
The name or ID of a previously installed task segment on the same program file as the overlay. The overlay is automatically deleted when the task segment is deleted.

### 6.8.5 Install Program Segment — IPS

The IPS command allows the user to install a segment on a program file and assign a segment ID. The IPS command and prompts are described below:

*Prompts:*

```
INSTALL PROGRAM SEGMENT
        PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
                SEGMENT NAME:  [alphanumeric]
                  SEGMENT ID:  [integer]
  OBJECT PATHNAME OR LUNO:     {pathname@/integer}      (*)
    DEFAULT SEGMENT FLAGS?:    YES/NO                   (YES)
```

If the response to the DEFAULT SEGMENT FLAGS?: prompt is NO, the following set of prompts is displayed on the user's terminal:

```
DEFINE SEGMENT FLAGS
                SYSTEM SEGMENT?:  YES/NO      (NO)
                MEMORY RESIDENT?: YES/NO      (NO)
                DELETE PROTECT?:  YES/NO      (NO)
                UPDATABLE?:       YES/NO      (NO)
                SHARABLE?:        YES/NO      (NO)
                REPLICATABLE?:    YES/NO      (NO)
            IN MEMORY REUSABLE?:  YES/NO      (NO)
            IN MEMORY COPYABLE?:  YES/NO      (NO)
```

After the responses are entered for the previous prompts, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
                EXECUTE PROTECT?:  YES/NO     (NO)
                WRITE PROTECT?:    YES/NO     (NO)
WRITABLE CONTROL STORAGE?:         YES/NO     (NO)
```

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of, or the LUNO assigned to, the program file on which the program segment is to be installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the IPS command. If zero is specified, the .S$SHARED program file is assumed.

SEGMENT NAME:
A user-defined character string, up to eight characters, composed of characters which are legal in pathnames. The segment name must be unique to the specified program file. If zero or a null response is specified, the IDT name of the object file will be used as the segment name.

SEGMENT ID:
A hexadecimal integer value that will be assigned as the ID of the program segment by the user. If zero or a null response is specified, the system will assign the ID.

OBJECT PATHNAME OR LUNO:
The pathname of, or the LUNO assigned to, the device or file where the object module for the program segment resides.

DEFAULT SEGMENT FLAGS?:
If YES is entered, the initial values are used for the flags. If NO is entered, the user has the option of which program segment flags will be modified.

SYSTEM SEGMENT?:
If YES is entered, the program segment may only be accessed by a system task.

MEMORY RESIDENT?:
If YES is entered, and the program segment is installed on the program file .S$SHARED or the applications program file specified at system generation, the program segment will be loaded into memory during initial program load (IPL) and remain in memory even when terminated.

DELETE PROTECT?:
If YES is entered, the program segment cannot be deleted from the program file unless the Modify Program Segment Entry (MSE) command is used to unprotect the program segment prior to the execution of the Delete Program Segment (DPS) command. If NO is specified, the program segment may be deleted by executing the DPS command.

UPDATABLE?:
If YES is entered, the data of a program segment may be modified, and the program segment will be written to disk with the new data modifications after the program segment is no longer used.

SHARABLE?:
If YES is entered, the program segment may be shared concurrently with more than one task.

REPLICATABLE?:
If YES is entered, there may be multiple copies of the program segment in memory simultaneously.

IN MEMORY REUSABLE?:
If YES is entered, the program segment in memory may be reused after termination by another task rather than a new copy being read from disk.

IN MEMORY COPYABLE?:
If YES is entered, the program segment may be copied from memory rather than being copied from disk. This situation may occur when the program segment is in memory and another user wishes to use the program segment.

EXECUTE PROTECT?:
If YES is entered, execution of the program segment is prohibited. The protection is enforced only on a 990/12 computer.

WRITE PROTECT?:
If YES is entered, the program segment may not be modified in memory. The protection is enforced only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the program segment uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

### 6.8.6 Delete Task — DT

The DT command removes a previously installed task from a program file. The task may be deleted by either name or by installed ID. If associated overlays exist, they are also deleted. The task may be specified by name or by installed ID, as shown below:

*Prompts:*

```
DELETE TASK SEGMENT
        PROGRAM FILE OR LUNO:  {filename@/integer}          (*)
              TASK NAME OR ID:  {alphanumeric/integer}
```

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name of, or LUNO assigned to, the program file on which the task segment has been installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the DT command.

TASK NAME OR ID:

The name or ID of the task segment on the specified program file.

### 6.8.7 Delete Procedure — DP

The DP command removes a previously installed procedure from a program file. The procedure may be specified by name or by installed ID, as shown below:

*Prompts:*

```
DELETE PROCEDURE SEGMENT
        PROGRAM FILE OR LUNO:  {filename@/integer}          (*)
        PROCEDURE NAME OR ID:  {alphanumeric/integer}
```

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name of, or the LUNO assigned to, the program file in which the procedure segment has been installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the DP command.

PROCEDURE NAME OR ID:

The name or ID of the procedure segment to be deleted from the specified program file.

### 6.8.8 Delete Overlay — DO

The DO command removes a previously installed overlay from a program file. The overlay may be specified by name or by installed ID, as shown below:

*Prompts:*

```
DELETE OVERLAY
        PROGRAM FILE OR LUNO:  {filename@/integer}          (*)
          OVERLAY NAME OR ID:  {alphanumeric/integer}
```

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of, or the LUNO assigned to, the program file on which the overlay has been installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the DO command.

OVERLAY NAME OR ID:
The name or ID of the overlay installed on the specified program file that is to be deleted.

### 6.8.9 Delete Program Segment — DPS

The DPS command is used to delete a segment from a specified program file. The program segment may be specified by name or be installed ID, as shown below:

*Prompts:*

```
DELETE PROGRAM SEGMENT
      PROGRAM FILE OR LUNO:  {filename@/integer}          (*)
        SEGMENT NAME OR ID:  {alphanumeric/integer}
```

*Prompt Details:*

PROGRAM FILE OR LUNO:
The filename of, or the LUNO assigned to, the program file on which the program segment has been installed. If a LUNO is specified in response to this prompt, it must be assigned to the program file prior to execution of the DPS command.

SEGMENT NAME OR ID:
The name or ID by which the program segment is known on the specified program file.

### 6.8.10 Modify Task Segment Entry — MTE

The MTE command allows the user to alter the data supplied when the task was installed. The values displayed are those defined during installation. Any of the displayed values may be changed, or the displayed value can be accepted by pressing the RETURN key. Refer to the IT command paragraph for the prompt descriptions for the task attributes. When the MTE command is called, the following appears:

*Prompts:*

```
MODIFY TASK SEGMENT ENTRY
   PROGRAM FILE PATHNAME:  filename@                   (*)
        MODULE NAME OR ID:  {alphanumeric/integer}
```

After the responses to the PROGRAM FILE PATHNAME: and MODULE NAME OR ID: prompts have been entered, the following set of prompts is displayed on the user's terminal:

```
MODIFY TASK ENTRY FOR ID <n>
                    NAME:  alphanumeric          (*)
              REAL TIME?:  YES/NO                (*)
                PRIORITY:  integer               (*)
           MODIFY FLAGS?:  YES/NO                (YES)
ATTACHED PROCEDURES?:  YES/NO                    (NO)
```

where <n> is the ID of the task to be modified. (This ID is for user information only and may not be modified.)

If YES was entered in response to the MODIFY FLAGS? prompt, the following prompts are displayed on the user's terminal:

```
MODIFY TASK FLAGS

                PRIVILEGED?:  YES/NO             (*)
              SYSTEM TASK?:  YES/NO              (*)
          MEMORY RESIDENT?:  YES/NO              (*)
             REPLICATABLE?:  YES/NO              (*)
           DELETE PROTECT?:  YES/NO              (*)
      IN MEMORY COPYABLE?:  YES/NO               (*)
      IN MEMORY REUSABLE?:  YES/NO               (*)
               UPDATABLE?:  YES/NO               (*)
    SOFTWARE PRIVILEGED?:  YES/NO                (*)
```

After the responses to the previous prompts are entered, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
          EXECUTE PROTECT?:  YES/NO              (*)
       OVERFLOW CHECKING?:  YES/NO               (*)
WRITABLE CONTROL STORAGE?:  YES/NO               (*)
```

If the response to the ATTACHED PROCEDURES?: prompt was YES, the following set of prompts is displayed on the user's terminal:

```
MODIFY TASK-ATTACHED PROCEDURES

          1ST PROCEDURE ID:  integer             (*)
P1 FROM TASKS PROGRAM FILE?:  YES/NO             (*)
          2ND PROCEDURE ID:  integer             (*)
P2 FROM TASKS PROGRAM FILE?:  YES/NO             (*)
```

*Prompt Details:*

PROGRAM FILE PATHNAME:
The file name of the program file on which the task segment to be modified has been installed.

MODULE NAME OR ID:
The task name or ID of the task segment installed on the specified program file.

NAME:
The name of the task. If the task ID was entered, the system automatically places the associated task name in the response field of this prompt.

REAL TIME?:
If YES is entered, the task segment to be modified was installed as a real-time task segment.

PRIORITY:
If YES was entered in response to the REAL TIME?: prompt, the priority value specified must be in the range of 1 through 127 (inclusive). If NO was entered, the priority value specified must be in the range of 0 through 4 (inclusive).

MODIFY FLAGS?:
If YES is entered, the user has the option of modifying the task flags.

PRIVILEGED?:
If YES is entered, the task is allowed to execute privileged hardware instructions. Privileged hardware instructions should be executed cautiously and only by the user who is very familiar with the system.

SYSTEM TASK?:
If YES Is entered, the task is allowed to execute in system memory space. For the task to be modified to become a system task, the task's load address must be greater than or equal to >C000. Tasks should be executed in system memory space with caution and only by the user who is very familiar with the system.

MEMORY RESIDENT?:
If YES is entered and the task is installed on the .S$SHARED program file or the applications program file specified at system generation, the task will be loaded into memory during initial program load (IPL) and remain in memory when terminated.

REPLICATABLE?:
If YES is entered, there may be multiple copies of the task in memory simultaneously.

DELETE PROTECT?:
If YES is entered, the task is protected against accidental deletion.

IN MEMORY COPYABLE?:
If YES is entered, the task segment may be copied from memory rather than being copied from disk. This situation may occur if the task is in memory and another user wishes to execute the task.

IN MEMORY REUSABLE?:
If YES is entered, the program segment in memory may be reused after termination by another task, rather than a new copy being copied from disk or being copied from one memory location to another.

UPDATABLE?:
If YES is entered, the data of a task may be modified, and the task segment will be written to disk with the new data modifications when the task terminates.

SOFTWARE PRIVILEGED?:
If YES Is entered, the task is allowed to execute privileged supervisor calls. Privileged supervisor calls should be executed with caution and only by the user who is very familiar with the system.

EXECUTE PROTECT?:
If YES is entered, execution of the task segment is prohibited. The protection is enforced only on a 990/12 computer.

OVERFLOW CHECKING?:
If YES is entered, the occurrence of arithmetic overflow will cause control of the task to pass to the task's end-action routine. Overflow checking is available only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the task uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

ATTACHED PROCEDURES?:
If YES is entered, the user has the option of modifying the procedures to be attached to the task segment.

1ST PROCEDURE ID:
The integer value representing the ID of a procedure segment attached to the task segment. If zero is entered, there are no procedure segments.

P1 FROM TASKS PROGRAM FILE?:
If YES is entered, the attached procedure segment with an ID specified for the 1ST PROCEDURE ID: prompt resides on the same program file as the task segment. If NO is entered, the procedure segment must reside on the .S$SHARED program file.

2ND PROCEDURE ID:
The integer value representing the ID of a procedure segment attached to the task segment. If zero is entered, there is no 2nd procedure segment.

P2 FROM TASKS PROGRAM FILE?:
If YES is entered, the attached procedure segment with an ID specified for the 2ND PROCEDURE ID: prompt resides on the same program file as the task segment. If NO is entered, the procedure segment must reside on the .S$SHARED program file.

### 6.8.11 Modify Procedure Entry — MPE

The MPE command allows the user to modify the data supplied when the procedure was installed. The values displayed are those defined when the procedure was installed. Any of the displayed values may be changed, or the displayed value can be accepted by pressing the RETURN key. When the MPE command is called, the following appears:

*Prompts:*

```
MODIFY PROCEDURE SEGMENT ENTRY
    PROGRAM FILE PATHNAME:  filename@                    (*)
        MODULE NAME OR ID:  {alphanumeric/integer}
```

After the responses to the PROGRAM FILE PATHNAME and MODULE NAME OR ID prompts have been entered, the following set of prompts are displayed on the user's terminal:

```
MODIFY PROCEDURE ENTRY FOR ID <n>
                      NAME:  alphanumeric                (*)
          MEMORY RESIDENT?:  YES/NO                      (*)
            DELETE PROTECT?: YES/NO                      (*)
```

where <n> is the ID of the procedure to be modified. (This ID is for user information only and may not be modified.)

After the responses to the above prompts are entered, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
             EXECUTE PROTECT?:  YES/NO                   (*)
               WRITE PROTECT?:  YES/NO                   (*)
WRITABLE CONTROL STORAGE?:      YES/NO                   (*)
```

*Prompt Details:*

PROGRAM FILE PATHNAME:

The file name of the program file where the procedure segment to be modified has been installed.

MODULE NAME OR ID:

The procedure name or ID of the procedure segment installed on the specified program file.

NAME:

The name of the procedure segment. If the procedure ID was entered, the system automatically places the associated procedure name in the response field of this prompt.

MEMORY RESIDENT?:

If YES is entered, and the procedure segment is installed on the program file .S$SHARED or the applications program file specified at system generation, the procedure segment will be loaded into memory during initial program load (IPL) and remain in memory even when terminated.

DELETE PROTECT?:
If YES is entered, the procedure segment is protected against accidental deletion.

EXECUTE PROTECT?:
If YES is entered, execution of the procedure segment is prohibited. The protection is enforced only on a 990/12 computer.

WRITE PROTECT?:
If YES is entered, procedure data cannot be modified in memory. The protection is enforced only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the procedure uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

### 6.8.12 Modify Overlay Entry — MOE

The MOE command allows the user to alter the data supplied when the overlay was installed. The values defined when the overlay was installed are displayed. Any of the entries may be changed, or the displayed value may be accepted by pressing the RETURN key. When the MOE command is called, the following appears:

*Prompts:*

```
MODIFY OVERLAY ENTRY
    PROGRAM FILE PATHNAME:  filename@                    (*)
         MODULE NAME OR ID:  {alphanumeric/integer}
```

After the responses to the previous prompts have been entered, the following prompts are displayed on the user's terminal:

```
MODIFY OVERLAY ENTRY FOR ID <n>
                      NAME:  alphanumeric                (*)
               RELOCATABLE?:  YES/NO                     (*)
           DELETE PROTECT?:  YES/NO                      (*)
```

where <n> is the ID of the overlay to be modified. (This ID is for user information only and may not be modified.)

*Prompt Details:*

PROGRAM FILE PATHNAME:
The file name of the program file on which the overlay is installed.

MODULE NAME OR ID:
The overlay name or ID of the overlay installed on the specified program file.

NAME:
The name of the overlay. If the overlay ID was entered, the system automatically places the associated overlay name in the response field of this prompt.

RELOCATABLE?:
If YES is entered, the overlay is allowed to be loaded at an address other than its natural load address.

DELETE PROTECT?:
If YES is entered, the overlay is protected against accidental deletion.

### 6.8.13 Modify Segment Entry — MSE

The MSE command allows the user to modify the attributes of a segment installed on a program file. The attribute values displayed are those defined when the segment was installed. Any of the displayed values may be changed, or the displayed values can be accepted by pressing the RETURN key. When the MSE command is called, the following appears:

*Prompts:*

```
MODIFY PROGRAM SEGMENT ENTRY
    PROGRAM FILE PATHNAME:  filename@                    (*)
        MODULE NAME OR ID:  {alphanumeric/integer}
```

After the responses to the PROGRAM FILE PATHNAME: and MODULE NAME OR ID: prompts have been entered, the following set of prompts is displayed on the user's terminal:

```
MODIFY PROGRAM SEGMENT ENTRY FOR ID <n>
                      NAME:  alphanumeric                (*)
            SYSTEM SEGMENT?:  YES/NO                     (*)
          MEMORY RESIDENT?:   YES/NO                     (*)
            DELETE PROTECT?:  YES/NO                     (*)
                UPDATABLE?:   YES/NO                     (*)
                 SHARABLE?:   YES/NO                     (*)
              REPLICATABLE?:  YES/NO                     (*)
      IN MEMORY REUSABLE?:    YES/NO                     (*)
      IN MEMORY COPYABLE?:    YES/NO                     (*)
```

where <n> is the ID of the segment to be modified. (This ID is for user information only and may not be modified.)

After the responses to the previous prompts are entered, the following prompts are displayed on the user's terminal:

```
990/12 FLAGS
                WRITE PROTECT?:  YES/NO                  (*)
              EXECUTE PROTECT?:  YES/NO                  (*)
WRITABLE CONTROL STORAGE?:       YES/NO                  (*)
```

*Prompt Details:*

PROGRAM FILE PATHNAME:
The file name of the program file on which the program segment to be modified has been installed.

MODULE NAME OR ID:
The segment name or ID of the program segment installed on the specified program file.

NAME:
The name of the program segment. If the segment ID was entered, the system automatically places the associated segment name in the response field of this prompt.

SYSTEM SEGMENT?:
If YES is entered, the program segment may only be accessed by a system task.

MEMORY RESIDENT?:
If YES is entered, and the program segment is installed on the program file .S$SHARED or the applications program file specified at system generation, the program segment will be loaded into memory during initial program load (IPL) and remain in memory even when terminated.

DELETE PROTECT?:
If YES is entered, the program segment is protected against accidental deletion.

UPDATABLE?:
If YES is entered, the data of a program segment may be modified, and the program segment will be written to disk with the new data modifications after the task is terminated or if the task maps the program segment out of its addressable memory area.

SHARABLE?:
If YES is entered, the program segment may be shared concurrently with more than one task.

REPLICATABLE?:
If YES is entered, there may be multiple copies of the program segment in memory simultaneously.

IN MEMORY REUSABLE?:
If YES is entered, the program segment in memory may be reused after termination by another task; rather than a new copy being copied from disk or being copied from one memory location to another.

IN MEMORY COPYABLE?:
If YES is entered, the program segment may be reused by copying the segment to more than one memory location rather than copying the segment from disk.

WRITE PROTECT?:
If YES is entered, the program segment may not be modified. The protection is enforced only on a 990/12 computer.

EXECUTE PROTECT?:
If YES is entered, execution of the program segment is prohibited. The protection is enforced only on a 990/12 computer.

WRITABLE CONTROL STORAGE?:
If YES is entered, the program segment uses the writable control storage area. Writable control storage is available only on a 990/12 computer.

## 6.9 INSTALLING IMAGE FORMAT WITH LINK EDITOR

The IMAGE format, selected by use of the FORMAT command, allows the Link Editor to install linked output memory images directly to a specified DNOS program file or to a DNOS image file. This feature allows the user to bypass the actual installation of tasks, procedures, and overlays. Linked output programs can replace existing programs or they can be added to the file. When the IMAGE format is selected and the overlays, tasks, and procedures are installed on a program file, the identifiers (IDs) of these overlays, tasks, and procedures are automatically assigned by the system. The assigned ID appears on the Load Map for the appropriate procedure, task, or phase. In order to load an overlay using a Load Overlay SVC, reference the overlay by name in the calling program, as shown:

```
REF overlay name

DATA overlay name
```

The Link Editor resolves the reference and stores the assigned overlay ID as the DATA statement operand. The ID may then be used in the supervisor call block.

**NOTE**

If the task name matches the overlay name, the task ID is stored in the DATA statement.

# 7

# Executing a Program

## 7.1 INTRODUCTION

Many commands are provided to execute tasks. Three of these commands are used for assembly language tasks, while the others are used for executing tasks of the various language processors available for the 990 computer.

## 7.2 EXECUTING AN ASSEMBLY LANGUAGE TASK

The three commands for executing assembly language tasks each serve a particular function. These commands are described and their syntax given in the following paragraphs.

### 7.2.1 Execute Task — XT

The XT command is used to execute a task and to leave SCI active during task execution. This command is used for most tasks, except those being debugged and terminal interactive tasks.

*Prompts:*

```
EXECUTE TASK
      PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
            TASK NAME OR ID:  {alphanumeric/integer}  (*)
                      PARM1:  integer                 (0)
                      PARM2:  integer                 (0)
                 STATION ID:  {integer/ME}            (*)
```

*Purpose:*

The XT command activates a program that does not interact with the user's terminal. Two 16-bit words of information can be passed to the program being activated in response to the PARM1 and PARM2 prompting messages of the XT command. The operating system automatically assigns a run-time ID to each program that it activates and displays the run-time ID at the user's terminal upon successful activation of the program. A task activated by the XT command cannot access event characters entered at the user's terminal.

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name of or the LUNO assigned to the program file on which the task to be executed has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the XT command. If zero is specified, the .S$SHARED program file is assumed.

TASK NAME OR ID:
Either the name or ID under which the program is installed on the specified program file.

PARM1: and PARM2:
Decimal or hexadecimal numbers in the range of 0 through 65535 representing a value to be passed to the program.

STATION ID:
The number (i.e., 2, not ST02) of the station with which the executing task is to be associated. A zero, or the characters ME indicates the user's terminal. A >FF indicates the task is not to be associated with a station.

A task should not be associated with a station unless it is used by the task for terminal I/O. If a station ID is specified through the XT command and SCI is quit (via the QUIT SCI command) before the task terminates, log on to SCI may not be performed until the task terminates.

### 7.2.2 Execute Task and Suspend SCI — XTS

The XTS command activates the specified task and suspends SCI until the task terminates. This command should be used for terminal interactive tasks to avoid contention between SCI and the task for terminal access.

*Prompts:*

```
EXECUTE TASK AND SUSPEND SCI
      PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
          TASK NAME OR ID:   {alphanumeric/integer}   (*)
                   PARM1:    integer                  (0)
                   PARM2:    integer                  (0)
              STATION ID:    {integer/ME}             (*)
```

*Purpose:*

The XTS command activates an interactive program and automatically suspends SCI to prevent it from interfering with the execution of the program. If SCI were not suspended, it would continue to interpret data entered at the terminal as though that data were intended for SCI, and an error would result. This command is also used to make event characters available to a task other than SCI.

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of or the LUNO assigned to the program file on which the task to be executed has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the XT command. If zero is specified, the .S$SHARED program file is assumed.

TASK NAME OR ID:
Either the name or ID under which the program is installed on the specified program file.

PARM1: and PARM2:
Decimal or hexadecimal numbers in the range of 0 through 65535 representing a value to be passed to the program.

STATION ID:
The number (i.e., 2, not ST02) of the station with which the executing task is to be associated. A zero, or the characters ME indicates the user's terminal. A >FF indicates the task is not to be associated with a station.

A task should not be associated with a station unless it is used by the task for terminal I/O. If a station ID is specified through the XT command and SCI is quit (via the QUIT SCI command) before the task terminates, log on to SCI may not be performed until the task terminates.

### 7.2.3 Execute and Halt Task — XHT

The XHT command places a task in memory in a suspended state so that it can be debugged. Typically, the user places the task to be debugged in memory using XHT, establishes the debug environment (including breakpoints), and then activates the task using the Resume Task (RT) command.

*Prompts:*

```
EXECUTE AND HALT TASK
      PROGRAM FILE OR LUNO:  {filename@/integer}      (*)
          TASK NAME OR ID:   {alphanumeric/integer}   (*)
                   PARM1:    integer                  (0)
                   PARM2:    integer                  (0)
               STATION ID:   {integer/ME}             (*)
```

*Purpose:*

The XHT command places a task in memory in a suspended state so that it can be debugged. Typically, the user places the task to be debugged in memory using XHT, establishes the debug environment (including breakpoints), and then activates the task using the Resume Task (RT) command.

*Prompt Details:*

PROGRAM FILE OR LUNO:
The file name of or the LUNO assigned to the program file on which the task has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the XHT command. If zero is specified, the .S$SHARED program file is used.

TASK NAME OR ID:
The name or the associated installed ID of the task whose execution is to be halted.

PARM1:
An integer value to be passed to the task being halted, determined by the programmer who wrote the task.

PARM2:
A second integer value to be passed to the task being halted, determined by the programmer who wrote the task.

STATION ID:
The station ID (e.g., 1, 2) with which the task is to be associated or the two-character pseudo device name of ME. If >FF is entered, the task is not associated with any station.

## 7.3 SVC EXECUTION OF TASK

The Execute Task supervisor call is used to initiate the execution of a task installed on any program file. If the task specified in the call is already active and was defined as being replicatable (during installation), another copy of the task is placed in execution. The replicated task can share procedures with previous activations of the task. If the call is issued for a task that is active but is not replicatable, the system returns an error to the calling task.

Refer to the *DNOS Supervisor Call (SVC) Reference Manual* for a complete description of the supervisor call block.

## 7.4 BATCH STREAM AND INTERACTIVE EXECUTION

Execution of an assembly language task may also be performed in a batch stream. The batch command stream for executing a task is depicted in Figure 7-1.

Refer to the section on assembling a program in this manual or the *DNOS System Command Interpreter (SCI) Reference Manual* for more information on batch stream operations.

```
BATCH
XT  PR=.USER.PF, T=TEST1, PARM1=0, PARM2=0, S=ME
EBATCH
```

**Figure 7-1. Execution Batch Stream**

# 8

# Debugging a Program

## 8.1 GENERAL INFORMATION

Flaws in software are commonly called "bugs". The process of removing flaws from software is called debugging. Modern programming techniques can drastically reduce the number of bugs in a program; however, the bugs which remain tend to be subtle and hard to find. DNOS provides several levels of debugging support, as follows:

- Several System Command Interpreter (SCI) commands provide debugging capabilities without requiring a special mode of operation.
- A special mode of operation allows a single task to be examined in detail during the execution process.

Since all of the debug commands interact with the terminal, special care must be taken when debugging a program that uses the terminal, because two processes requesting terminal support can be confusing to the user. If the program being debugged requests use of a terminal, two terminals should be used: one for executing the program and one for debugging.

## 8.2 MODES OF DEBUGGING

There are two sets of debug commands:

- Commands used for debugging all tasks.
- Controlled task commands used for tasks that have been put into the debug mode through the use of the Execute Debug (XD) command.

**NOTE**

Putting a task into debug mode affects the execution of all debug commands as follows:

— Symbolic expressions may be used in place of integer expressions as responses to commands involving a controlled task.

— Every command functions as if the controlled task is unconditionally suspended.

— Every command leaves the controlled task unconditionally suspended.

— Tasks which unconditionally suspend themselves can be momentarily reactivated by some of the debug commands.

— The CMD key automatically suspends the controlled task when executing the Proceed from Breakpoint (PB), Delete and Proceed from Breakpoint (DPB), or Resume Task (RT) commands.

### 8.2.1 Unconditional Suspend

Most of the debugging commands require that the task being debugged be unconditionally suspended either before or during the debug command. The "unconditional suspend" task state under DNOS (task state 6) is the state in which the task is dormant until activated by a command. There are several ways for a task to become unconditionally suspended:

- The task is bid with the suspend option selected. Either a supervisor call, the Execute and Halt Task (XHT) command, or the .DBID SCI primitive suspend a task when the task is bid.

  The XHT command is used for tasks normally executed by an Execute Task (XT) command. XHT places the task in a suspended state for debugging and displays the run ID of the task to the user. If the user desires to execute and halt the task, and simultaneously place it in controlled mode, the Execute Debug (XD) command may be used with no input for the RUN ID prompt. The XD command performs the XHT and saves the run ID as the default for the Debugger commands.

  The .DBID primitive is used for tasks that interface through SCI, such as command processors which are normally bid using the .BID and .QBID primitives, described in the *DNOS System Command Interpreter (SCI) Reference Manual.* When the .DBID primitive is executed through SCI, the task is bid and immediately placed in a suspended state. The run ID of the task is saved in the synonym $$BT or it may be obtained by issuing a Show Task Status (STS) command.

- The task suspends itself.

- The task executes a breakpoint (XOP 15,15).

- The task is suspended by the SCI debug commands.

Once the task has been placed in a suspended state, the Debugger may be used to assign breakpoints, simulate execution, display memory, and perform other debugging functions. When the debugging session is over, the task may be terminated by the Kill Task (KT) command. If the task was put into controlled mode by an XD command, it may be killed by responding YES to the KILL TASK? prompt of the Quit Debug (QD) command.

### 8.2.2 Symbols

The debug support provided allows symbolic debugging; whereby, the user can specify labels within the task being debugged rather than memory addresses. This method of debugging is both convenient and meaningful since the source code list can be used as reference for the symbolic

labels used. Symbolic constants consist of the link edit phase name, a period (.), the module identifier name (IDT), a period (.), and the symbol, an assembly language label. The syntax is defined as:

phase name.IDT name.symbol

**NOTE**

To have full symbolic capability, both the assembler and Link Editor must have used the SYMT option.

If the assembler did not use the SYMT option, but the Link Editor did, then symbols of the following form are available:

phase name.IDT name

If either the phase name or the IDT name of a symbol is omitted, the immediately preceding phase name or IDT name is used. The syntax is as follows:

.IDT name.symbol (no phase name)

phase name..symbol (no IDT name)

..symbol (no phase or IDT name)

*Examples:*

| | | | |
|---|---|---|---|
| PHASE1.MOD1.XYZ | References Phase | = | PHASE1 |
| | IDT | = | MOD1 |
| | Label | = | XYZ |
| .MOD2.MNO | References Phase | = | PHASE1 |
| | IDT | = | MOD2 |
| | Label | = | MNO |
| ..ABC | References Phase | = | PHASE1 |
| | IDT | = | MOD2 |
| | Label | = | ABC |

Four words of memory per symbol are required to store symbol values.

If the task being debugged is a single routine installed without being linked, then the symbolic constant consists of a period (.), the characters of the module identifier name, a period (.), and the characters of the symbol, as follows:

.IDT name.symbol

**NOTE**

Symbols may only be used for commands affecting a task that has been placed in the debug mode by the Execute Debug (XD) command.

Symbol encoding uses a hashing method which sometimes produces a seeming duplication of values for a symbol. In such cases, use another symbol.

### 8.2.3 Expressions

Constants (and symbolic constants for tasks in the debug mode) may be combined using the operators +, -, *, /, ( ), and < > to form expressions which may be used as command operands. The operators have the following meanings:

| Operator | Meaning |
|---|---|
| + | Unary plus or addition |
| - | Unary minus or subtraction |
| * | Multiplication |
| / | Division |
| ( ) | Evaluation order |
| < > | Indicated memory location contents |

**NOTE**

The right angle bracket, >, is regarded as a hexadecimal number indicator rather than the right part of < > whenever there are hexadecimal digits immediately following. Thus, no conflict arises.

Expressions are evaluated according to the following rules:

- Subexpressions delimited by ( ) and < > are evaluated first with the innermost expression evaluated before any other levels.
- Unless otherwise instructed by parentheses or angle brackets, unary + and - are evaluated first, multiplication and division are evaluated second, and addition and subtraction last.
- For operators at the same level, evaluation proceeds left to right.
- Arithmetic treats all constants as unsigned numbers.

For example, if .IDTNAM.BEGIN is memory address >7A, and if memory address >7F contains >3B, then the expression FF/(IDTNAM.BEGIN + 5 + -2 + 3*>F) is evaluated as follows:

```
>FF/(<.IDTNAM.BEGIN + 5> + -2 + 3*>F)
>FF/(<>7A + 5> + -2 + 3*>F)
>FF/(<>7F> + -2 + 3*>F)
>FF/(>3B + -2 + >2D)
>FF/(>3B + (-2) + >2D)
>FF/(>39 + >2D)
>FF/>66
2
```

These symbols may be used in expression lists in the same way as constants or symbolic constants. For example,

```
#PC  +  NAME.IDT  -  #R15
```

is a valid expression.

Several special symbols are allowed in expressions. These special symbols are:

| Symbol | Description |
|---|---|
| #PC | Contents of the Program Counter |
| #WP | Contents of the Workspace Pointer |
| #ST | Contents of the Status Register |
| #Rn | Contents of the Workspace Register whose number corresponds to the number (0 through 15) given for n. |

Character strings are also allowed in expressions. A character string is of the form 'XXXX__' where X is any valid ASCII character. The apostrophe can be represented in a character string by using double apostrophes. A character string may be any length, but only the leftmost four characters are significant. Strings shorter than four characters are right-justified with leading zeros. The value of a character string is an expression in the ASCII hexadecimal representation of the characters expressed as a 32-bit number.

| String | Value |
|---|---|
| 'ABCD' | 41424344 |
| 'A' | 00000041 |
| 'ABCDE' | 41424244 |
| ' ' | 00000020 |
| 'A''B' | 00412742 |

## 8.3 COMMANDS FOR ALL TASKS

The SCI commands described in the following paragraphs may be used for all tasks. These commands are most frequently used in debugging; however, may be used whenever SCI is active.

Many of the debug commands require the run-time task ID returned by the XT or XHT commands. Make note of the run-time task ID when the task is placed in execution. The Show Task Status (STS) command may be used to identify the run-time ID (which identifies the task to DNOS).

### 8.3.1 Data Display Commands

These SCI commands display the contents of memory, registers and specified breakpoint addresses. Table 8-1 lists the paragraphs associated with each command for easier referencing.

**Table 8-1. Debug Commands**

| Debug Command | Paragraph Reference |
|---|---|
| Activate Task | 8.3.4.1 |
| Assign Breakpoints | 8.3.3.1 |
| Assign Simulated Breakpoints | 8.3.6.1 |
| Delete and Proceed From Breakpoint | 8.3.3.3 |
| Delete Breakpoints | 8.3.3.2 |
| Delete Simulated Breakpoints | 8.3.6.2 |
| Execute and Halt Task | 8.3.4.5 |
| Find Byte | 8.3.5.1 |
| Find Word | 8.3.5.2 |
| Halt Task | 8.3.4.2 |
| Initiate Debug Mode | 8.3.4.4 |
| List Breakpoints | 8.3.1.1 |
| List Logical Record | 8.3.1.2 |
| List Memory | 8.3.1.3 |
| List Simulated Breakpoints | 8.3.6.3 |
| List System Memory | 8.3.1.4 |
| Modify Absolute Disk | 8.3.2.1 |
| Modify Allocatable Disk Unit | 8.3.2.2 |
| Modify Internal Registers | 8.3.2.3 |
| Modify Memory | 8.3.2.4 |
| Modify Program Image | 8.3.2.5 |
| Modify Relative to File | 8.3.2.6 |
| Modify System Memory | 8.3.2.7 |
| Modify Workspace Registers | 8.3.2.8 |
| Proceed from Breakpoint | 8.3.3.4 |
| Quit Debug Mode | 8.3.6.4 |
| Resume Simulated Task | 8.3.6.5 |
| Resume Task | 8.3.4.3 |
| Show Absolute Disk | 8.3.1.5 |
| Show Allocatable Disk Unit | 8.3.1.6 |
| Show Internal Registers | 8.3.1.7 |
| Show Panel | 8.3.1.8 |
| Show Program Image | 8.3.1.9 |
| Show Relative to File | 8.3.1.10 |
| Show Value | 8.3.1.11 |
| Show Workspace Registers | 8.3.1.12 |
| Simulate Task | 8.3.6.6 |

**8.3.1.1 List Breakpoints — LB.** The LB command is used for displaying the breakpoints for a specified task. If the breakpoints are to be displayed for a system task, the user must have a privileged user ID.

*Prompts:*

```
LIST BREAKPOINTS
                    RUN ID:  integer                (*)
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.1.2 List Logical Record — LLR.** The LLR command lists the contents of a record or records in a file. The contents of the record or records specified are listed in both hexadecimal and ASCII representation. The amount displayed per record is a maximum of decimal 512 (hexadecimal 200) or the logical record length of the file, whichever is less.

*Prompts:*

```
LIST LOGICAL RECORD
                      PATHNAME:  pathname@          (*)
                STARTING RECORD:  integer           (0)
              NUMBER OF RECORDS:  [integer]
          LISTING ACCESS NAME:  [pathname@]
```

*Prompt Details:*

PATHNAME:
The pathname that identifies the file in which the records to be listed reside.

STARTING RECORD:
A decimal or hexadecimal integer that identifies the first record whose contents are to be listed.

NUMBER OF RECORDS:
A decimal or hexadecimal integer that identifies how many records are to be listed. A null response specifies that all records are to be listed.

LISTING ACCESS NAME:
The device name of a device or the pathname of a file to which the LLR command should write the contents of the record(s) specified. The default value is the terminal local file.

**8.3.1.3 List Memory — LM.** The LM command is used to list the specified memory area of a task to a specified output device or file. If the task is not unconditionally suspended, it is temporarily suspended while the listing is being formatted.

*Prompts:*

```
LIST MEMORY
                    RUN ID:  integer                    (*)
          STARTING ADDRESS:  full exp
           NUMBER OF BYTES:  [full exp]
     LISTING ACCESS NAME:  [pathname@]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

STARTING ADDRESS:
The integer value which is the starting address of the memory area to be listed.

NUMBER OF BYTES:
The integer value which is the number of bytes of memory to be listed, beginning with the specified starting address. The default value is 16 bytes.

LISTING ACCESS NAME:
The device name or file name of the device or file where the memory list is to be output. The default value is the terminal local file.

**8.3.1.4 List System Memory — LSM.** The LSM command is used to list the memory occupied by the DNOS operating system. This command is similar to the List Memory (LM) command, except the user specifies an overlay name or ID instead of a run ID.

The LSM command is intended for use only by someone very familiar with DNOS source code.

*Prompts:*

```
LIST SYSTEM MEMORY
        OVERLAY NAME OR ID:  {integer/alphanumeric}
          STARTING ADDRESS:  integer
           NUMBER OF BYTES:  [integer]
     LISTING ACCESS NAME:  [pathname@]
```

*Prompt Details:*

OVERLAY NAME OR ID:
The overlay name or integer value specified in the Install Overlay (IO) command which is the ID of the overlay whose memory is to be listed. By executing the Map Program File (MPF) command on the kernel program file, (whose name is specified at system generation, the user may inspect the acceptable overlay names and associated IDs.

STARTING ADDRESS:
The integer expression which is the starting address of the memory area to be listed.

NUMBER OF BYTES:
The integer value which is the number of bytes of memory to be listed, beginning with the specified starting address. The initial value is >40 bytes.

LISTING ACCESS NAME:
The device name or file name where the memory list is to be output. The default value is the terminal local file.

**8.3.1.5 Show Absolute Disk — SAD.** The SAD command is used to print the contents of a specified absolute address on a disk and may be executed only by privileged users. The contents of sixteen bytes are printed per line, with the address of the first byte printed as the first entry on the line. The contents of each pair of bytes are shown as four hexadecimal digits. At the right end of the line, the contents are printed as ASCII characters. The bytes that contain values that correspond to printable ASCII characters are translated and printed as ASCII characters; nonprinting ASCII characters are printed as periods.

*Prompts:*

```
SHOW ABSOLUTE DISK
                    DISK UNIT:  devicename@
                         TRACK:  integer exp
                        SECTOR:  integer exp
                    FIRST WORD:  integer exp              (0)
                NUMBER OF WORDS:  [integer exp]
            OUTPUT ACCESS NAME:  [pathname@]              (*)
```

*Prompt Details:*

DISK UNIT:
The device name assigned to the disk during system generation. Normally, the characters DS01 are used for the system disk and DSxx for other disks on the system; where xx is a two digit decimal number less than or equal to ten (for example DS02).

TRACK:
The integer value that is the starting track address from which to begin printing the contents of the disk.

SECTOR:
The integer value that is the starting sector address, within the specified disk track, from which to begin printing the contents of the disk.

FIRST WORD:
The integer value that is the word offset, within the specified disk sector, from which to begin printing the contents of the disk.

NUMBER OF WORDS:
The integer value that is the number of words of the specified sector to print. The default value is the disk sector size.

OUTPUT ACCESS NAME:
The device name or file name of a device or file where the contents of the specified absolute disk address is to be printed. The default value is the terminal local file.

**8.3.1.6 Show Allocatable Disk Unit — SADU.** The SADU command is used to output the contents of the specified allocatable disk units (ADUs) to the specified device.

All disks on a DNOS system are addressed in ADUs, the basic addressable disk unit in a DNOS system. The maximum number of ADUs on a disk is 65,535. Therefore, if a disk contains more than 65,535 sectors, multiple sectors are used as ADUs.

*Prompts:*

```
SHOW ALLOCATABLE DISK UNIT
                DISK UNIT:  devicename@
              ADU NUMBER:   integer exp
            SECTOR OFFSET:  integer exp
               FIRST WORD:  integer          (0)
         NUMBER OF WORDS:   [integer exp]
      OUTPUT ACCESS NAME:   [pathname@]      (*)
```

*Prompt Details:*

DISK UNIT:
The device name assigned to the disk during system generation. Normally, the characters DS01 are used for the system disk and DSxx for other disks on the system, where xx is a two-digit decimal number greater than one (e.g., DS02).

ADU NUMBER:
The integer value that is the ADU with contents to be listed.

SECTOR OFFSET:
The integer value that is the sector of the ADU with contents to be listed.

FIRST WORD:
The integer value that is the word offset, within the specified sector, from which to begin listing the contents of the ADU.

NUMBER OF WORDS:
The integer value that is the number of words of the specified sector to list. The default value is the disk ADU size.

OUTPUT ACCESS NAME:
The device or file name where the contents of the specified ADU are to be listed. The default value is the terminal local file.

**8.3.1.7 Show Internal Registers — SIR.** The SIR command is used to display the task state and the contents of the internal registers of a task: program counter (PC), workspace pointer (WP), workspace register (WR), status register (ST), memory, and breakpoints. The STATE field is the

state of the task before it was suspended to show the contents of the internal registers. The remainder of the display reflects the internal register values in effect after the task was suspended.

The character string representation of the status register follows the hexadecimal value and may include the following characters:

| | |
|---|---|
| L = Logical greater than | P = Parity |
| A = Arithmetic greater than | X = XOP in progress |
| E = Equal | S = Privileged mode |
| C = Carry | M = Map file |
| O = Overflow | |

If the internal registers are to be shown for a system task, the user must have a privileged user ID.

*Prompts:*

```
SHOW INTERNAL REGISTERS
                    RUN ID:  integer                (*)
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.1.8 Show Panel — SP.** The SP command is used to display the debug panel for a specified task. If the task is not unconditionally suspended, it will be temporarily suspended while the panel is being formatted and displayed. The displayed task state is the state of the task before it was suspended. The debug panel consists of the following:

- Internal registers
- Workspace registers
- Breakpoints
- Memory display
- Task state

The SP command also shows the character string representation of the status register.

If the debug panel to be displayed is for a system task, the user must have a privileged user ID.

*Prompts:*

```
SHOW PANEL
                    RUN ID:  integer                (*)
          MEMORY ADDRESS:  [full exp]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

MEMORY ADDRESS:
The integer value that is the starting memory address for the memory portion of the debug panel display. The default value is the current PC address.

**8.3.1.9 Show Program Image — SPI.** The SPI command is used to display the disk-resident memory image of a module (defined as a task, procedure, segment, or overlay) for a specified program.

*Prompts:*

SHOW PROGRAM IMAGE

| | | |
|---|---|---|
| PROGRAM FILE: | filename@ | (*) |
| OUTPUT ACCESS NAME: | [pathname@] | (*) |
| MODULE TYPE: | {T/P/O/S} | (*) |
| MODULE NAME OR ID: | {alphanumeric/integer} | (*) |
| ADDRESS: | integer | (*) |
| LENGTH: | integer | (040) |

*Prompt Details:*

PROGRAM FILE:
The file name of or the LUNO assigned to the program file on which the program (task, procedure, overlay, or segment) has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the SPI command. If zero is specified, the .S$SHARED program file is assumed.

OUTPUT ACCESS NAME:
The device name or file name where the display of the memory image of the program is to be written. The default value is the terminal local file.

MODULE TYPE:
The type of program with a memory image to be displayed. The following characters are valid responses:

T = Task
P = Procedure
O = Overlay
S = Program Segment

MODULE NAME OR ID:
The characters or the associated ID that identifies the program on the specified program file.

ADDRESS:
The integer value that is the starting address of the memory image to be displayed.

LENGTH:
The integer value that is the number of words of the memory image to be displayed.

**8.3.1.10 Show Relative to File — SRF.** The SRF command is used to display any word or group of words within a file. It assumes that the user has knowledge of the file structure and allows the user to address any word within the file.

*Prompts:*

```
SHOW RELATIVE TO FILE
                    PATHNAME:  filename@                (*)
               RECORD NUMBER:  integer                  (*)
                  FIRST WORD:  integer                  (*)
            NUMBER OF WORDS:   [integer]
          OUTPUT ACCESS NAME:  [pathname@]              (*)
```

*Prompt Details:*

PATHNAME:
The name of the file with a record to be displayed.

RECORD NUMBER:
The integer value that is the record number within the file to be displayed.

FIRST WORD:
The integer value that is the byte offset within the record to be displayed.

NUMBER OF WORDS:
The integer value that is the number of words of the record to display. The default is to display the whole record.

OUTPUT ACCESS NAME:
The pathname of a device or file where the results of the SRF command are to be listed. The default is the terminal local file.

**8.3.1.11 Show Value — SV.** The SV command is used to display the value of a specified expression. The hexadecimal, decimal, and ASCII representations of the value are given.

*Prompts:*

```
SHOW VALUE
                    EXPRESSION:  full exp
```

*Prompt Details:*

EXPRESSION:
The integer and/or character(s) expression with a value to be displayed. If a task is being debugged and is a controlled task, the expression may be symbolic.

**8.3.1.12 Show Workspace Registers — SWR.** The SWR command is used to display the current workspace of a task. If the task is not unconditionally suspended, it is temporarily suspended while the workspace is displayed.

If the terminal requesting the command is a VDT, the SWR command functions the same as the Show Panel (SP) command. If the workspace to be displayed is for a system task, the user must have a privileged user ID.

*Prompts:*

```
SHOW WORKSPACE REGISTERS
                    RUN ID:  integer                (*)
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

### 8.3.2 Data Modification Commands

These commands are used to place specified data on a disk or change data at an absolute word address. Modification of specified ADUs, internal registers, memory image, or programs may be accomplished using these debugging commands.

**8.3.2.1 Modify Absolute Disk — MAD.** The MAD command is used to place specified data on a disk at a specified absolute track, sector, and word address and may only be executed by privileged users. Data is entered in groups of word values to be placed on disk. Word values must be separated from each other with a comma and loaded on disk in successive addresses. The verification parameter allows the user to enter a string of words to be compared to the data at the specified address. If there is not a correspondence between the string of words and the data at the specified address, the modification does not take place.

**NOTE**

Since the MAD command has the capability to write anything, anywhere on the disk, and can therefore destroy the DNOS system image, the verify option should always be used.

*Prompts:*

```
MODIFY ABSOLUTE DISK
                  DISK UNIT:  devicename@
        OUTPUT ACCESS NAME:  [pathname@]          (*)
                     TRACK:  integer exp
                    SECTOR:  integer exp
                FIRST WORD:  integer exp
         VERIFICATION DATA:  [integer(s)]
                      DATA:  integer(s)
```

*Prompt Details:*

DISK UNIT:
The device name of the disk device assigned during system generation. Normally, the characters DS01 are used for the system disk and DS0x for other disks on the system, where x is a digit greater than one.

OUTPUT ACCESS NAME:
The device or file name where the contents of the specified absolute disk address are to be printed. The default value is the terminal local file.

TRACK:
The integer value which is the starting track address from which to begin the disk modification.

SECTOR:
The integer value which is the starting sector address, within the specified disk track, from which to begin the disk modification.

FIRST WORD:
The integer value which is the starting word address, within the specified disk sector, from which to begin the disk modification.

VERIFICATION DATA:
If specified, the integer value contained in the specified starting address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive words, beginning with the specified first word.

DATA:
The integer value to replace the existing value contained in the specified first word. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive words, beginning with the first word.

**8.3.2.2 Modify Allocatable Disk Unit — MADU.** The MADU command is used to modify a specified allocatable disk unit (ADU). If verification data does not match the data already on the disk, modification will not be performed.

All disks on a DNOS system are addressed in ADUs. The maximum number of ADUs on a disk is 65,535. Therefore, if a disk contains more than 65,535 sectors, multiple sectors are used as ADUs. ADUs are the basic addressable disk unit in a DNOS system.

*Prompts:*

```
MODIFY ALLOCATABLE DISK UNIT
                    DISK UNIT:  devicename@
          OUTPUT ACCESS NAME:   [pathname@]                (*)
                 ADU NUMBER:    integer exp
              SECTOR OFFSET:    integer exp
                 FIRST WORD:    integer exp
           VERIFICATION DATA:   [integer exp list]
                        DATA:   integer exp list
```

*Prompt Details:*

DISK UNIT:
The device name of the disk assigned during system generation. Normally, the characters DS01 are used for the system disk and DS0x for other disks on the system, where x is a digit greater than one.

OUTPUT ACCESS NAME:
The device or file name where the results of the ADU modification are to be listed. The default value is the terminal local file.

ADU NUMBER:
The integer value which is the ADU with contents to be modified.

SECTOR OFFSET:
The integer value which is the sector of the ADU with contents to be modified.

FIRST WORD:
The integer value which is the starting word offset, within the specified sector, where modifications of the ADU are to begin.

VERIFICATION DATA:
If specified, the integer value contained in the specified first word address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive words, beginning with the specified first word.

DATA:
The integer value to replace the existing value contained in the specified first word. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive words, beginning with the first word.

**8.3.2.3 Modify Internal Registers — MIR.** The MIR command is used to modify the internal registers of a task: program counter (PC), workspace pointer (WP), and status register (ST). If the task being debugged is not a privileged task, then only bits 0 through 6 of the status register can be modified with this command. If the task is not unconditionally suspended, it is temporarily suspended while the command is interacting with the register modification.

As in the Modify Memory (MM) command, the MIR command is interactive; the RETURN key may be pressed after the register and its contents have been displayed and/or modified to cause the next register and its contents to be displayed. Also, by pressing the Command (CMD) key, SCI is returned to command mode.

If the internal registers to be modified are for a system task, the user must have a privileged user ID.

*Prompts:*

```
MODIFY INTERNAL REGISTERS
                    RUN ID:   integer                    (*)
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.2.4 Modify Memory — MM.** The MM command is used to modify the memory image of a task, starting at the address specified. If the task is not unconditionally suspended, it is temporarily suspended while the command is interacting. Swapping does not affect the modification process. Consecutive memory addresses, and their values, may be displayed and/or modified by pressing the RETURN key. Pressing the Command (CMD) key will return SCI to command mode.

*Prompts:*

```
MODIFY MEMORY
                    RUN ID:  integer                (*)
                 ADDRESS:  full exp
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

ADDRESS:
The integer value of the first memory address to be modified.

**8.3.2.5 Modify Program Image — MPI.** The MPI command is used to modify a program (defined to be a task, procedure, or overlay) in a specified program file.

*Prompts:*

```
MODIFY PROGRAM IMAGE
                PROGRAM FILE:  filename@                (*)
          OUTPUT ACCESS NAME:  [pathname@]              (*)
                 MODULE TYPE:  {T/P/O/S}                (*)
          MODULE NAME OR ID:  {alphanumeric/integer}    (*)
                    ADDRESS:  integer                   (*)
           VERIFICATION DATA:  [integer(s)]
                        DATA:  integer(s)
                   CHECKSUM:  [integer(s)]
          RELOCATION OF DATA?:  [YES/NO...YES/NO]
```

*Prompt Details:*

PROGRAM FILE:
The file name of or the LUNO assigned to the program file on which the program (task, procedure, segment, or overlay) to be modified has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the MPI command. If zero is specified, the .S$SHARED program file is assumed.

OUTPUT ACCESS NAME:
The device name or file name where the results of the memory image modification of the program are to be written. If a null response is specified, the terminal local file is used.

MODULE TYPE:
The type of program with a memory image to be modified. The following characters are valid responses:

T = Task
P = Procedure
O = Overlay
S = Program Segment

MODULE NAME OR ID:
The character(s) or the associated ID which identifies the program on the specified program file.

ADDRESS:
The integer value which is the starting address of the memory image to be modified.

VERIFICATION DATA:
If specified, the integer value contained in the specified starting address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in consecutive memory addresses, beginning with the specified starting address.

DATA:
The integer value to replace the existing value contained in the specified starting address. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in consecutive memory addresses, beginning with the specified starting address.

CHECKSUM:
The checksum is an exclusive OR of each word of new data. If the checksum is not known and a null response is entered, the checksum will be printed to the device or file specified in response to the OUTPUT ACCESS NAME: prompts.

RELOCATION OF DATA?:
If YES is specified, the data value will be relocated when the task is loaded into memory for execution. NO specifies that the data value will not be relocated. If a list of data values are specified in response to the DATA prompt and relocation is desired, the user must specify which values are to be relocated. That is, a YES or NO response must be entered for each corresponding data value. If there is a list of YES or NO responses, they must be separated by commas.

**8.3.2.6 Modify Relative to File — MRF.** The MRF command changes data at an absolute word address within a file. It is assumed that the user has knowledge of the file and disk structure. Addresses above 64K (65,536) bytes must have a record number and sector offset supplied by the user. Words below 64K bytes can be addressed directly and the sector is located by the program. Verification should be used, when possible.

*Prompts:*

```
MODIFY RELATIVE TO FILE
                    PATHNAME:  filename@                (*)
          OUTPUT ACCESS NAME:  [pathname@]              (*)
               RECORD NUMBER:  integer                  (*)
                  FIRST WORD:  integer                  (*)
           VERIFICATION DATA:  [integer...integer]
                        DATA:  integer...integer
                    CHECKSUM:  [integer]
```

*Prompt Details:*

PATHNAME:
The file name with contents to be modified.

OUTPUT ACCESS NAME:
The device name or file name where the results of the MRF command will be listed. If a null response is specified, the terminal local file is used.

RECORD NUMBER:
The integer value which is the physical record number within the file to be modified. If the specified word address is over 64K bytes, the user must supply the sector offset as the response to this prompt.

FIRST WORD:
The integer value which is the starting byte offset where the modification of the record is to begin. The byte offset must be on an even boundary.

VERIFICATION DATA:
If specified, the integer value contained in the specified first word address. If more than one integer is specified, they must be separated by commas; it is assumed these values are contained in successive word addresses, beginning with the specified first word address.

DATA:
The integer value to replace the existing value contained in the specified first word address. If more than one value is specified, they must be separated by commas; it is assumed these values are to replace the existing values contained in successive word addresses, beginning with the specified first word address.

CHECKSUM:
The checksum is an exclusive OR of each word of new data. If the checksum is not known and a null response is entered, the checksum will be printed to the device or file specified in response to the OUTPUT ACCESS NAME: prompt.

**8.3.3.3 Delete and Proceed from Breakpoint — DPB.** The DPB is used to proceed from a brea point at which a task is currently stopped and to delete that breakpoint. If the breakpoint h already been deleted, the command functions as if it were a Proceed from Breakpoint (F command.

*Prompts:*

```
DELETE AND PROCEED FROM BREAKPOINT
                    RUN ID:  integer                    (*)
DESTINATION ADDRESS(ES):  [full exp list]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Shc Task Status (STS) command.

DESTINATION ADDRESS(ES):
The integer value(s) of the address(es) within the task which are additional breakpoir to be set. A null response specifies that no new breakpoints are to be set.

**8.3.3.4 Proceed from Breakpoint — PB.** The PB command is used to resume execution of task that is stopped at a breakpoint without deleting the breakpoint. The task is resume executing the instruction at the breakpoint at which it is currently stopped; however, the brea point remains active. If the task is not currently at a breakpoint, the user is notified by a warni message that the task is not at a breakpoint, and the task remains in whatever state it was before the PB command.

The PB command may also be used to assign new breakpoints in the specified task by respondi to the DESTINATION ADDRESS(ES) prompt. Breakpoints are set, if possible, at all the specifi destination addresses. If no destination address(es) is specified, the task resumes execution b no breakpoints are set.

*Prompts:*

```
PROCEED FROM BREAKPOINT
                    RUN ID:  integer                    (*)
DESTINATION ADDRESS(ES):  [full exp list]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Shc Task Status (STS) command.

DESTINATION ADDRESS(ES):
The integer value(s) of the address(es) within the task where the new breakpoints are occur. Addresses must be separated by a comma. The default value is no ne breakpoints.

### 8.3.4 Task Control Commands

The control commands are used to unconditionally suspend and activate the task during the debugging process.

**8.3.4.1 Activate Task — AT.** The AT command is used to activate an unconditionally suspended task.

*Prompts:*

ACTIVATE TASK

RUN ID: integer (*)

*Prompt Details:*

RUN ID:
A valid task run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.4.2 Halt Task — HT.** The HT command is used to unconditionally suspend a task at the end of the current time slice. If the task is already unconditionally suspended, it has no effect on the task. If the task is not in the active state, the HT command waits five seconds for the task to reach unconditional suspend, then gives the user the option of aborting or continuing to wait. This option occurs every five seconds if the task is not active and the HT command is executed.

If the task cannot be suspended, the following message is displayed:

UNABLE TO SUSPEND TASK. CURRENT STATE$^{5}$XX. CONTINUE COMMAND?

If a YES response is entered, another attempt is made to suspend the task. If unsuccessful, the message is displayed again. A NO response to the preceding message causes the following message to be displayed:

DO YOU WISH TO LEAVE SUSPENSION PENDING?

A YES response leaves the suspension pending, while a NO response terminates the suspension attempt.

If the specified task is a system task, the user must have a privileged user ID.

*Prompts:*

HALT TASK

RUN ID: integer (*)

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.4.3 Resume Task — RT.** The RT command is used to activate a task at the point at which it was suspended. The specified task must be unconditionally suspended when this command is executed or an error is indicated. The Delete Breakpoint (DB) and the RT command, Delete and Proceed from Breakpoint (DPB) or Proceed from Breakpoint (PB) commands must be used to restart a task halted at a breakpoint. The RT command should be used instead of the Activate Task (AT) command, to reactivate a task halted by the Halt Task (HT) command.

*Prompts:*

RESUME TASK

| | | |
|---|---|---|
| RUN ID: | integer | (*) |

*Prompt Details:*

RUN ID:
The response to this prompt must be a valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

**8.3.4.4 Execute in Debug Mode — XD.** The XD command is used to place a specified task into controlled mode. The run-time ID is optional but cannot be the ID of a system task. If no run-time ID is specified, an automatic call is made to the Execute and Halt Task (XHT) command to place the task into execution.

The symbol table object file is optional and its presence determines whether symbolic expressions are allowed on any of the subsequent debug commands. If a symbol table was specified to the Link Editor (SYMT option was selected) and if the controlled task symbol table object file is specified, then symbolic expressions involving symbols in the object code symbol table may be used in commands that call for string parameters.

The debugger may be used to simulate 990 computer object code. The command defaults to the object code of the host computer.

Only one task for each station may be in debug mode at a given time.

*Prompts:*

EXECUTE IN DEBUG MODE

| | | |
|---|---|---|
| RUN ID: | [integer] | (*) |
| SYMBOL TABLE OBJECT FILE: | [filename@] | (*) |
| 990/12 OBJECT CODE?: | YES/NO | (YES) |

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

If a null response is specified, the prompts for the Execute and Halt Task (XHT) command are displayed. Refer to the XHT command for information concerning responses to these prompts.

SYMBOL TABLE OBJECT FILE:

The file name specified to the Link Editor if the SYMT option has been selected. By specifying this file name in response to the prompt, the user is allowed to use symbolic expressions which involve symbols in the object code symbol table on any debug command prompt which calls for a character(s) response. If a null response is entered, no symbol table file is used and symbolic expressions are not allowed.

990/12 OBJECT CODE?:

If YES is entered in response to this prompt, the debugger will simulate 990/12 object code if executing on a 990/12 computer. If NO is entered, the debugger will simulate 990/10 object code whether executing on a 990/10 or 990/12 computer.

**8.3.4.5 Execute and Halt Task — XHT.** The XHT command is used to place a task in memory in a suspended state so that it can be debugged. Typically, the user places the task to be debugged in memory using XHT, establishes the debug environment (including breakpoints), and then activates the task using the Resume Task (RT) command.

*Prompts:*

EXECUTE AND HALT TASK

| | | |
|---|---|---|
| PROGRAM FILE OR LUNO: | {filename@/integer} | (*) |
| TASK NAME OR ID: | {alphanumeric/integer} | (*) |
| PARM1: | integer | (0) |
| PARM2: | integer | (0) |
| STATION ID: | {integer/ME} | (*) |

*Prompt Details:*

PROGRAM FILE OR LUNO:

The file name of or the LUNO assigned to the program file on which the task has been installed. If a LUNO is specified in response to this prompt, it must be assigned prior to the execution of the XHT command. If zero is specified, the .S$SHARED program file is used.

TASK NAME OR ID:

The name or the associated installed ID of the task whose execution is to be halted.

PARM1:

An integer value to be passed to the task being halted, determined by the programmer who wrote the task.

PARM2:

A second integer value to be passed to the task being halted, determined by the programmer who wrote the task.

STATION ID:

The station ID (e.g., 1, 2) with which the task is to be associated or the two-character pseudo device name of ME. If >FF is entered, the task is not associated with any station.

### 8.3.5 Search Commands

The search commands are used to search for the specified value(s) in a memory area of a task.

**8.3.5.1 Find Byte — FB.** The FB command is used to search for the specified value(s) in a memory area of a task; with the search beginning on a byte boundary. If the specified value is found, the corresponding memory address is displayed. If the task is not unconditionally suspended, it is temporarily suspended while the search is performed.

*Prompts:*

```
FIND BYTE
                    RUN ID:  integer                (*)
                  VALUE(S):  full exp list
         STARTING ADDRESS:   [full exp]
           ENDING ADDRESS:   [full exp]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

VALUE(S):
The integer value(s) to find in the memory area of the task.

STARTING ADDRESS:
The integer value which is the starting address of the memory area to be searched. The default is zero.

ENDING ADDRESS:
The integer value which is the ending address of the memory area to be searched. The default is end of task.

**8.3.5.2 Find Word — FW.** The FW command is used to search for the specified value(s) in a memory area of a task; with the search beginning on a word boundary. If the specified value is found, the corresponding memory address is displayed. If the task is not unconditionally suspended, it is temporally suspended while the search is performed.

*Prompts:*

```
FIND WORD
                    RUN ID:  integer                (*)
                  VALUE(S):  full exp list
         STARTING ADDRESS:   [full exp]
           ENDING ADDRESS:   [full exp]
```

*Prompt Details:*

RUN ID:
A valid run ID in the user's job. Current run IDs may be obtained by executing the Show Task Status (STS) command.

VALUE(S):
The integer value(s) to find in the memory area of the task.

STARTING ADDRESS:
The integer value which is the starting address of the memory area to be searched. If an odd address is specified, the address is rounded up to the nearest even value. The default is zero.

ENDING ADDRESS:
The integer value which is the ending address of the memory area to be searched. The default address is the end of task.

### 8.3.6 Controlled Task Commands

The control commands allow control and trace execution of instructions in a task until:

- The execution of a specified number of instructions has been simulated.
- A specified address is placed in the PC.
- A breakpoint or simulated breakpoint occurs.

**8.3.6.1 Assign Simulated Breakpoint — ASB.** The ASB command is used to set up a breakpoint on a range of values for memory as follows:

- Memory alteration (A)
- CRU access (C)
- Program Counter value (P)
- Memory references (R)
- Status register value (S)

A memory write operation, which does not change the value in memory, is not a memory alteration. The breakpoints set with this command are only valid during a Simulate command. Breakpoints, in this case, are conditions which stop execution but allow execution to be resumed by an operator command, either by using the Resume Simulated Task (RST) command or by pressing the F3 function key. Each simulated breakpoint is assigned a number which is displayed at the completion of the ASB command. When a breakpoint occurs during simulation, a panel and the breakpoint number are displayed along with the display string.

*Prompts:*

```
ASSIGN SIMULATED BREAKPOINT
          ON (A,C,P,R,S):  {A/C/P/R/S}          (PC)
                   FROM:   full exp
                   THRU:   [full exp]
                  COUNT:   full exp              (1)
                DISPLAY:   [full exp]
```

*Prompt Details:*

ON (A,C,P,R,S):
The characters A, C, P, R, S are valid responses to this prompt and have the following meanings:

A = Memory alteration
C = CRU access
P = Program Counter value
R = Reference (memory)
S = Status Register value

FROM:
The integer expression that specifies the lower address limit for breakpointing.

THRU:
The integer expression that specifies the upper address limit for breakpointing. The default value is the value specified for the FROM: prompt.

COUNT:
The integer expression that specifies the number of times this breakpoint is to be encountered before execution is halted. The default value is one.

DISPLAY:
The integer expression that specifies the memory address to be displayed when this breakpoint is reached. The default value is the PC value at the time the breakpoint is reached.

**8.3.6.2 Delete Simulated Breakpoints — DSB.** The DSB command is used to allow the user to delete a list of simulated breakpoints assigned with the Assign Simulated Breakpoint (ASB) command.

*Prompts:*

DELETE SIMULATED BREAKPOINTS
BREAKPOINT NUMBERS: [full exp list/ALL]

*Prompt Details:*

BREAKPOINT NUMBERS:
The integer value that specifies the number of the breakpoint to delete, which was the breakpoint number returned by the ASB command. If the characters ALL are entered, all the simulated breakpoints are deleted. The default is the breakpoint at which the task is stopped. Current simulated breakpoints may be obtained by executing the List Simulated Breakpoints (LSB) command.

**8.3.6.3 List Simulated Breakpoints — LSB.** The LSB command is used to display all current simulated breakpoints. When the breakpoints are listed, the first column of the display lists the numbers assigned when the breakpoints were set; the numbers start at one and are consecutive. The TYPE column lists letters for the ON prompt of an Assign Simulated Breakpoint command to identify the value on which the breakpoint was set, and the FROM and THRU columns list the

corresponding operand addresses. The COUNT column lists the count operand entered when the breakpoints were set, and the REMAINING column lists the number of times the program has yet to go through the breakpoint. The DISPLAY column lists the display operand.

When the operands represent CRU addresses or ST register values, the operands are listed as hexadecimal numbers.

*Prompts:*

None

**8.3.6.4 Quit Debug Mode — QD.** The QD command is used to take a controlled task out of debug mode. The user has the option of killing the task at this point. If the user chooses not to kill the task, it will be left unconditionally suspended; but the user may still issue any of the general SCI commands. The Resume Task (RT) or Proceed from Breakpoint (PB) commands (depending on whether the task is at a breakpoint) may be used to activate the task.

The RT command is discussed in the task control commands paragraphs.

*Prompts:*

QUIT DEBUG MODE

KILL TASK ?: YES/NO (YES)

*Prompt Details:*

KILL TASK?:

If YES is entered, the current executing task will be killed. The task then executes its end-action routine. If NO is entered, the current executing task will be unconditionally suspended.

**8.3.6.5 Resume Simulated Task — RST.** The RST command is used to allow the user to resume simulation following a breakpoint, a simulated breakpoint, or simulation of a specified number of instructions. The last entered values for the FOR: and TO: prompts of the Simulate Task (ST) command are used as the RST limits. Upon reaching a terminating condition (breakpoint, simulated breakpoint, time-out or the value specified for the TO: prompt), a panel and termination reason are displayed. Simulation may be continued by pressing the F3 function key or terminated by pressing the Command (CMD) key, which returns SCI to the command mode.

**8.3.6.6 Simulate Task — ST.** The ST command is used to provide controlled and traced execution of the instructions in a task. Controlled execution continues until the execution of a specified number of instructions has been simulated or until a specified address is placed in the PC or until a breakpoint or simulated breakpoint occurs. Simulation may be continued by pressing the F3 function key.

Simulated execution continues without operator intervention and locks out further SCI commands. Following simulation of the instruction whose address is specified by the response to the TO: prompt, SCI displays the panel and halts simulation. The user can regain SCI capabilities by pressing the Command (CMD) key to return to command mode.

When the number of specified simulations has been performed, SCI displays the following message and halts simulation:

TIME OUT

*Prompts:*

SIMULATE TASK

| | | |
|---|---|---|
| FOR: | [full exp] | (*) |
| FROM: | [full exp] | |
| TO: | [full exp] | |

*Prompt Details:*

FOR:
The integer expression that specifies the number of instruction simulations to be performed and must be less than or equal to 32,767. When the specified number of simulations has been performed, SCI displays the following message and halts simulation:

TIME OUT

If a null response is entered for this prompt, the value specified in a previous ST command is used; if no previous ST commands were executed, a one is used.

FROM:
The integer expression that specifies the address of the first instruction to be simulated. If a null response is entered in response to this prompt, simulation begins at the instruction with an address in the PC.

TO:
The integer expression that specifies the address of the last instruction to be simulated. The integer expression entered may be less than that entered for the FROM command. If a null response is entered in response to this prompt, simulation continues until a breakpoint or simulated breakpoint is encountered or until the user presses the CMD key, returning SCI to command mode.

Messages:

STOP AT TRAP NO. X

where X is the number of the simulated breakpoint set through the Assign Simulated Breakpoint (ASB) command.

## 8.4 STATION DEPENDENT DISPLAYS

As mentioned previously, the displays generated by debugging SCI commands vary in format and content depending on the display device. High-speed display terminals (such as Video Display Terminals) display more information than slower, hard copy terminals. Table 8-2 lists the display generated by several of the debug commands in varying environments.

**Table 8-2. Command Displays**

| Command | Hard Copy Regular | Hard Copy Debug | VDT Regular | VDT Debug |
|---|---|---|---|---|
| AB | — | — | — | PANEL |
| DB | — | — | — | PANEL |
| PB | — | — | — | PANEL |
| DBP | — | — | — | PANEL |
| LB | BRKPTS | BRKPTS | BRKPTS | BRKPTS |
| HT | — | — | — | PANEL |
| RT | — | — | — | PANEL |
| MM | INTERACT | INTERACT | INTERACT | INTERACT PANEL |
| LM | TLF | TLF | LF | TLF |
| FW | MSG OR TLF | MSG OR TLF | MSG OR TLF | MSG + PANEL OR TLF |
| FB | MSG OR TLF | MSG OR TLF | MSG OR TLF | MSG + PANEL OR TLF |
| SIR | INT REG | INT REG | PANEL | PANEL |
| MWR | INTERACT | INTERACT | INTERACT | INTERACT PANEL |
| SWR | WKSPC | WKSPC | PANEL | PANEL |
| SP | PANEL | PANEL | PANEL | PANEL |
| SV | VALUES | VALUES | VALUES | VALUES |
| XD | — | — | — | PANEL |
| ASB | — | — | — | BRKPT NO. + PANEL |
| DSB | — | — | — | PANEL |
| LSB | — | SIMULATED BRKPTS | — | SIMULATED BRKPTS |
| ST | — | TRAP#OR'TIMEOUT' | — | TRAP#OR'TIMEOUT' + PANEL |
| RST | — | TRAP#OR'TIMEOUT' | — | TRAP#OR'TIMEOUT' + PANEL |
| QD | — | — | — | — |

BRKPTS = Breakpoints
INTERACT = Interactive
INT REG = Internal registers
MSG = Message
PANEL = Debug panel
TIMEOUT = Time-out
TLF = Terminal local file
TRAP#OR = Trap number
WKSPC = Workspace

# 9

# Assembly Language Program Example

## 9.1 EXAMPLE PROGRAMMING

This paragraph describes a simple procedure for creating and executing an assembly language program using DNOS. This brief program is assembled with the SYMT command entered in the OPTIONS?: prompt of the Execute Macro Assembler (XMA) command.

This program may be assembler without the Symbol Table by omitting the SYMT OPTION in the XMA command; however, symbolic debugging cannot be performed without the Symbol Table.

The program is debugged in three different ways:

- Symbolic Debugging — The Symbol Table is supplied to the linked object making the addresses of the labels in the program recognizable to the debugger.

- Breakpoint Debugging — Breakpoints are assigned to addresses in the executed program. When a breakpoint is reached, execution halts, and the panel is displayed, showing the address values of the breakpoint.

- Simulated Debugging — Used in the same example as the symbolic debugging. The address values are displayed for the address range specified during the Assign Simulated Breakpoints (ASB) command until a breakpoint or end of execution is reached.

The examples also explain how to write messages to the terminal, or to an assigned file by assigning a Logical Unit Number (LUNO) to the terminal or file.

An example is supplied, near the end of the section, describing the execution of a previously debugged program.

For more detailed information on how to code a program, consult the *Assembly Language Reference Manual.*

The brief assembly language example given in this section displays a message and requests the input of three numbers. The program, for this example, was used as the Text Editor Example (Figure 4-1) and entered in file .USER.SOURCE.

The procedures given in this section are for use on a 911 VDT.

The directory .USER, previously created in the section on building a program, is used to simplify file references during assembly and execution. A suggested syntax is as follows:

| | |
|---|---|
| Source file: | .USER.SOURCE |
| Object file: | .USER.OBJECT |
| Listing file: | .USER.LISTING |
| Link edit listing file: | .USER.LNKLIST |
| Linked output file: | .USER.LNKOUT |
| Error file: | .USER.ERROR |
| Message file: | .USER.MESSAGE |
| Link edit control file: | .USER.CNTRLINK |

The volume name is optional if the system disk (DS01) is used, and it may be omitted.

The .USER.SOURCE file is already created. The remaining files, except .USER.MESSAGE are created automatically by the following procedures. It is necessary to create the file .USER.MESSAGE, using the Create File Sequential (CFSEQ) command, as shown below:

```
[ ] CFSEQ

CREATE SEQUENTIAL FILE
                    PATHNAME:  .USER.MESSAGE
       LOGICAL RECORD LENGTH:  <Press RETURN>
    PHYSICAL RECORD LENGTH:  <Press RETURN>
          INITIAL ALLOCATION:  <Press RETURN>
       SECONDARY ALLOCATION:  <Press RETURN>
             EXPANDABLE?:  YES
          BLANK SURPRESS?:  NO
            FORCED WRITE?:  NO
```

The messages produced by the program are written to this file instead of the terminal, since SCI has command of the terminal.

## 9.2 REVIEW OF TEXT EDITING

A quick review on entering the program into the computer is discussed in this paragraph.

1. Power up the computer and terminal and log-on using the procedures given in Section 2.

2. Invoke the Text Editor by entering the Execute Editor (XE) command. The following parameter appears:

   ```
   [ ] XE

   EXECUTE TEXT EDITOR
     FILE ACCESS NAME:  <Press TAB key>
   ```

3. Press the unlabeled gray key or the RETURN key to create the first blank line above the *EOF record.

4. Type in the program source code.

5. Press CMD, after entering the source code, to leave the compose mode.

6. Enter the Quit Editor (QE) command to quit the Text Editor. Select the following parameters:

```
[ ] QE

QUIT EDIT
                    ABORT?:   NO

QUIT EDIT
OUTPUT FILE ACCESS NAME:   .USER.SOURCE
                  REPLACE?:   NO
     MOD LIST ACCESS NAME:   <Press RETURN>
```

## 9.3 ASSEMBLE THE PROGRAM

1. Invoke the macro assembler by entering XMA command and select the following parameters:

```
[ ] XMA

 EXECUTE MACRO ASSEMBLER
SOURCE ACCESS NAME:   .USER.SOURCE
OBJECT ACCESS NAME:   .USER.OBJECT
LISTING ACCESS NAME:  .USER.LISTING
 ERROR ACCESS NAME:   .USER.ERROR
           OPTIONS:   SYMT
MACRO LIBRARY PATHNAME: <Press RETURN>
  PRINT WIDTH (CHARS): 80
 PAGE LENGTH (LINES):  60
```

2. Enter the Wait command.

```
[ ] WAIT

        —WAITING FOR BACKGROUND TASK TO COMPLETE—
```

3. When the assembly completes, the following message is displayed:

```
I ASSEMBLER-0001 MACRO ASSEMBLY COMPLETE, 0000 ERROR(S) 0000
WARNING(S)
```

4. Press the RETURN key to return to the command mode.

## 9.4 LINK EDIT THE OBJECT CODE

1. First create a command file for the Link Editor. Invoke the Text Editor by entering XE command. Press the TAB key to clear the display.

```
[ ] XE

 INITIATE TEXT EDITOR
   FILE ACCESS NAME:   <Press TAB key>
```

2. Place the Text Editor in compose mode by pressing F7 and then press the unlabeled gray key for the first blank line above the EOF* record.

3. Enter the following lines into the control file:

```
TASK TEST
INCLUDE .USER.OBJECT
END
```

4. Leave the compose mode by pressing the CMD key.

5. Quit the Text Editor by entering QE. Select the following parameters:

```
[ ] QE

QUIT EDIT
                ABORT?:   NO

QUIT EDIT
OUTPUT FILE ACCESS NAME:    .USER.CNTRLINK
                 REPLACE?:  N
     MOD LIST ACCESS NAME:  <Press RETURN>
```

6. Invoke the Link Editor by entering the Execute Link Editor (XLE) command. Select the following parameters:

```
[ ] XLE

     EXECUTE LINK EDITOR
  CONTROL ACCESS NAME:   .USER.CNTRLINK
LINKED OUTPUT ACCESS NAME:  .USER.LNKOUT
      LISTING ACCESS NAME:  .USER.LNKLIST
       PRINT WIDTH (CHARS):  80 <Press RETURN>
```

7. The SCI prompt [ ] appears, enter the WAIT command and press RETURN key. The following display appears:

```
[ ] WAIT

    —WAITING FOR BACKGROUND TASK TO COMPLETE—
```

When the Link Editor terminates, the following is displayed:

```
I LINKER-0001 LINK EDITOR COMPLETED, 0 ERROR(S), 0 WARNING(S)
```

8. Press the CMD key to return to command mode.

## 9.5 INSTALL THE PROGRAM

The program must now be installed as a DNOS task by use of the Install Task (IT) command. A program file is required for the IT command. The .USER.PROGA program file created in Section 6, can be used in this example. Perform the following steps to install the task:

1. Enter the IT command to place the program on .USER.PROGA program file. Specify the following parameters:

```
[ ] IT

   INSTALL TASK SEGMENT
PROGRAM FILE OR LUNO:  .USER.PROGA
          TASK NAME:   TEST
            TASK ID:   0
OBJECT PATHNAME OR LUNO: .USER.LNKOUT
          PRIORITY:    4
DEFAULT TASK FLAGS?:   YES
ATTACHED PROCEDURES:   NO
```

   The installed ID is displayed in the following form when the installation is completed:

```
TASK NAME  =  TEST
TASK ID    =  >run-time ID
```

2. Press the CMD key to return to the command mode.

3. The program uses LUNO >20 and the LUNO must be assigned to either the VDT or the file .USER.MESSAGE. For the first examples the LUNO is assigned to .USER.MESSAGE. Call the Assign Luno (AL) command and respond as follows:

```
[ ] AL

ASSIGN LUNO

          LUNO:  >20
   ACCESS NAME:  .USER.MESSAGE
PROGRAM FILE?:   NO
```

   The message ASSIGNED LUNO: >20 is then displayed.

4. Press the CMD key to return to the command mode.

## 9.6 EXECUTE THE PROGRAM — SYMBOLIC DEBUGGING WITH SIMULATION

Symbolic debugging involves the use of the Symbol Table. During the XMA command, the OPTIONS: SYMT must be entered to include the Symbol Table in the object code. Inclusion of the Symbol Table allows you to reference addresses by the label name rather than the address on any SCI command where an address is required, such as SP and AB. An example of the object code containing the Symbol Table information is shown in Figure 9-1.

Notice the tag character, address, and label are presented at the bottom of the code. For example,

G012ECLOSE

To execute the program:

1. Use the Execute and Halt Task (XHT) command. Use of this command activates the task but does not begin execution. When XHT is entered, the following prompt is displayed. Respond as shown:

```
[] XHT

        EXECUTE AND HALT TASK
PROGRAM FILE NAME OR LUNO:  .USER.PROGA
          TASK NAME OR ID:  TEST
                   PARM1:   0
                   PARM2:   0
              STATION ID:   ME
```

   The following message appears:

```
RUNTIME TASK ID  =  >run-time ID
```

2. Note that the run-time ID of the task is returned on the display. Remember the run-time ID for the next step. Return to the command mode.

```
0015CRESPONSEA0000C0006C013CB0000A0006A0026B0000B0020B0000B00007F211F    RESP0001
B0000B0000B0000B0B20B0000C003EBC000B004AB0A0DB4845B4C4CB4F2CB20507F1CFF RESP0002
B4C45B4153B4520B494EB5055B5420B4E55B4D42B4552B204FB4620B4954B454D7F187F RESP0003
B5320B534FB4C44B2054B4F44B4159B2E20B2055B5345B2034B2D44B4947B49547F199F RESP0004
B204EB554DB4245B5253B2E00A0086B0A0DB0000B0B20B0040C00A8B0000B000A7F1C8F RESP0005
C0096C009CB0004B0000A009CA00A8B0A0DB4954B454DB2031B2020B0000B0B207F1D1F RESP0006
B0040C00C6B0000B000AC00C0C00A0B0004B0000B0A0DB4954B454DB2032B20207F1EFF RESP0007
B0000B0B20B0040C00E4B0000B000AC00DEC00A2B0004B0000B0A0DB4954B454D7F1CEF RESP0008
B2033B2020B2000A00F0B0000B0B20B0000C00FCB0000B0032B0A0DB5448B414E7F1F1F RESP0009
B4B20B594FB5520B464FB5220B594FB5552B2050B5552B4348B4153B452EB20487F197F RESP0010
B4156B4520B4120B4E49B4345B2044B4159B2E00A012CB0A0DB0000B0120B00007F1E6F RESP0011
B0000B0000B0000B0400B2FE0C0026B2FE0C0032B2FE0C0088B2FE0C00B2B2FE07F195F RESP0012
C00D0B2FE0C00F0B2FE0C012EB2FE0C013A7F7D9F                                RESP0013
G012ECLOSE G013AEOP   G00FCGOODBYG003EGREET G00A8ITEM1 7F26AF             RESP0014
G00C6ITEM2 G00E4ITEM3 G0032MSSG0 G0088MSSG1 G00B2MSSG2 7F2A7F             RESP0015
G00D0MSSG3 G00F0MSSG4 G0026OPEN  2013CG013CSTART G009CSTORE 7F150F        RESP0016
G0096STR1  G00C0STR2  G00DESTR3  G0006WSP   7F5C1F                       RESP0017
:       RESPONSE                                                         RESP0018
```

**Figure 9-1. Object Code with Symbol Table**

3. Place the task in the debug mode by entering the Execute Debug (XD) command. Respond to the following prompts (answer NO to the 990/12 OBJECT CODE, if using 990/10 system.), as shown:

```
[ ] XD

EXECUTE IN DEBUG MODE
                RUN ID:  >run-time ID
SYMBOL TABLE OBJECT FILE:  .USER.LNKOUT
     990/12 OBJECT CODE?:  YES
```

4. The contents of the panel appear.

5. Assimulated breakpoints may be assigned to aid in debugging by entering the ASB command, as follows:

```
[ ]ASB

ASSIGN SIMULATED BREAKPOINT
        ON (A,C,P,R,S):  PC (default)
                FROM:  >14E
                THRU:  >14E
               COUNT:  1
             DISPLAY:
```

   The panel display and message SIMULATED BREAKPOINT 1 appear. Press the RETURN key.

6. To begin execution of the task, use the Simulate Task (ST) command. The prompts and responses are as follows:

```
[ ] ST

SIMULATE TASK

                FOR:  100
               FROM:  <Press RETURN>
                 TO:  <Press RETURN>
```

   The panel display for address >14E appears on the screen, with the message STOP AT TRAP #1. Press the CMD key to return to the command mode.

7. To exit the debug mode, enter the Quit Debug (QD) command and respond as below:

```
[ ] QD

QUIT DEBUG MODE
        KILL TASK?:  NO <Press RETURN>
```

8. To resume execution from the breakpoint enter the Resume Task (RT) command, as shown below:

```
[ ] RT

RESUME TASK
                RUN ID:   >run-time ID <Press RETURN>
```

9. The test program has executed. Perform a Show File (SF) command on the file .USER.MESSAGE. The following messages appear in the file.

```
GOOD MORNING, PLEASE INPUT NUMBER OF ITEMS SOLD TODAY. USE 4-DIGIT
NUMBERS.

ITEM 1
ITEM 2
ITEM 3
THANK YOU FOR YOUR PURCHASE. HAVE A NICE DAY.
```

Press the CMD key to return to the command mode.

10. Execute the Release Luno (RL) command, to release LUNO >20 assigned to .USER.MESSAGE, as shown below:

```
[ ] RL

RELEASE LUNO
                LUNO:   >20
```

11. Delete file .USER.MESSAGE with the Delete File (DF) command, as shown below:

```
[ ] DF

DELETE FILE
        PATHNAME(S):   .USER.MESSAGE
```

## 9.7 EXECUTE THE PROGRAM — BREAKPOINT DEBUGGING

In this example, the RESPONSE program is debugged using assigned breakpoints, and the responses are written to USER.MESSAGE file.

1. Create the .USER.MESSAGE file and assign LUNO >20 to the file, as shown above.

2. To execute the task, use the XHT command. Use of this command activates the task but does not begin execution. The XHT command is useful when the debugging commands are to be used for the task. When XHT is entered, the following prompt is displayed. Respond as shown:

```
[ ] XHT

        EXECUTE AND HALT TASK
PROGRAM FILE NAME OR LUNO:  .USER.PROGA
          TASK NAME OR ID:  TEST
                   PARM1:   0
                   PARM2:   0
              STATION ID:   ME
```

   The following message appears:

```
RUNTIME TASK ID  =  >run-time ID
```

3. Note that the run-time ID of the task is returned on the display. Remember the run-time ID for the next step. Return to the command mode.

4. To assign breakpoints and stop execution of the task at location >146, enter the Assign Breakpoints (AB) command. Respond to the following prompts as shown:

```
[ ] AB

ASSIGN BREAKPOINTS
              RUN ID:  >run-time ID
        ADDRESS(ES):   >146
```

   The panel showing the address values of the WORKSPACE REGISTERS, BREAKPOINTS (0146), and MEMORY appear on the screen. A panel display similar to the one in Figure 9-2 appears:

```
RUN ID=FD  STATE=06     WP=0006      PC=0142     <PC>=2FE0     ST=018F          M
                      W O R K S P A C E   R E G I S T E R S

0006  0000  0000  0000  0000  0000  0000  0000  0000  .. .. .. .. .. .. .. ..
0016  0000  0000  0000  0000  0000  0000  0000  0000  .. .. .. .. .. .. .. ..
                        B R E A K P O I N T S
0146

                                  M E M O R Y
0142  2FE0  0026  2FCF  0032  2FE0  008E  2FE0  00B8  /. .& /. .2 /. .. /. ..
0152  2FE0  00D6  2FE0  00F6  2FE0  0134  2FE0  0140  /. .. /. .. /. .4 /. .@
0162  C000  0008  0000  0000  0026  0001  3002  6002  .. .. .. .. .& .. 0. \.
```

**Figure 9-2. Panel Display**

5. Enter the RT command to reach the specified assigned breakpoint in the program. The following appears on the screen:

```
[ ] RT

RESUME TASK
                RUN ID:   >run-time ID <Press RETURN>
```

6. Task execution begins.

7. To display memory contents of the assigned breakpoint, the Show Panel (SP) command or the Show Internal Registers (SIR) is entered. In this example the SP command is used. Respond to the prompts as follows:

```
[ ] SP

SHOW PANEL
                RUN ID:   >run-time ID
    MEMORY ADDRESS:   >146
```

   The panel values for address >146 appear on the screen in the section MEMORY.

8. To resume execution of the task, the Proceed from Breakpoint (PB) command must be entered. The following prompts appear:

```
[ ] PB

PROCEED FROM BREAKPOINT
                RUN ID:   >run-time ID
DESTINATION ADDRESS(ES):  <Press RETURN>
```

9. Execution is complete. View the .USER.MESSAGE file with the SF command. The following appears in the file:

```
GOOD MORNING, PLEASE INPUT NUMBER OF ITEMS SOLD TODAY. USE 4-DIGIT
NUMBERS.

ITEM 1
ITEM 2
ITEM 3
THANK YOU FOR YOUR PURCHASES. HAVE A NICE DAY.
```

10. Press the RETURN key to enter the command mode.

11. Execute the RL command to release LUNO >20 assigned to .USER.MESSAGE and delete the file.

## 9.8 EXECUTE THE PROGRAM — NO DEBUGGING

This method is used when the task has previously been debugged and is ready to execute.

1. Assign LUNO >20 to the terminal with the AL command, as shown below:

```
[ ] AL

ASSIGN LUNO
                LUNO:  >20
         ACCESS NAME:  ME
       PROGRAM FILE?:  NO
```

   The message ASSIGNED LUNO: >20 appears. Return to the command mode.

2. Execute the program using the Execute Task and Suspend SCI (XTS) command. Select the following parameters:

```
[ ] XTS

  EXECUTE TASK AND SUSPEND SCI
PROGRAM FILE OR LUNO:  .USER.PROGA
      TASK NAME OR ID:  TEST
                PARM1:  0
                PARM2:  0
           STATION ID:  ME
```

3. The test program now executes. The greeting and ITEM 1 appear. Enter a four-digit number in response. The next ITEM # appears after the four-digit response is complete. Enter another 4-digit number for all ITEM # prompts (1 through 3). The closing message appears. Below is an example of what appears:

```
= = FOREGROUND COMMAND EXECUTING = =
GOOD MORNING, PLEASE INPUT NUMBER OF ITEMS SOLD TODAY. USE 4-DIGIT
NUMBERS.

ITEM 1 2571
ITEM 2 3123
ITEM 3 4619
THANK YOU FOR YOUR PURCHASE. HAVE A NICE DAY.
```

4. After the closing message, the RUNTIME TASK ID = >run-time ID appears on the screen.

5. Press the CMD key to return to the initial SCI menu.

6. Delete the task entry by using the Delete Task (DT) command as follows:

```
[ ] DT

  DELETE TASK
PROGRAM FILE OR LUNO:  .USER.PROGA
      TASK NAME OR ID:  TEST
```

## 9.9 DELETE DIRECTORY

To delete the directory .USER from the system disk, enter the Delete Directory (DD) command, as shown below:

```
[ ] DD

DELETE DIRECTORY
            PATHNAME:  .USER
LISTING ACCESS NAME:   <Press RETURN>
        ARE YOU SURE:  YES
```

The directory created and all files in it are now deleted. Return the terminal to the command mode.

# Appendix A

# Abnormal Completion Messages

The following messages are issued by the assembler upon abnormal completion of processing. In addition to these messages, a number of messages are issued to the user in the assembly listing file and/or in the file specified in response to the ERROR ACCESS NAME prompt of the XMA procedure.

The codes listed below are defined in the *DNOS Messages and Codes Reference Manual.*

**Message**

SOURCE FILE I/O ERROR, CODE = XXXX
OBJECT FILE I/O ERROR, CODE = XXXX
LIST FILE I.O ERROR, CODE = XXXX
TEMP FILE I.O ERROR, CODE = XXXX

The messages listed below are assembler bugs. If the message reappears on subsequent assemblies, load a fresh copy of the assembler from a backup disk. If the error still persists, contact your customer representative.

**Assembler Bugs**

ATTEMPT TO POP EMPTY STACK — SDSMAC BUG
DIRECTIVE EXPECTED — SDSMAC BUG
UNEXPECTED END OF PARSE — SDSMAC BUG
ERROR MAPPING PARSE — SDSMAC BUG
INVALID OPERATION ENCOUNTERED — SDSMAC BUG
NO OP CODE — SDSMAC BUG
INVALID LISTING ERROR ENCOUNTERED
SYMBOL TABLE ERROR
MACRO EXPANSION ERROR
BUG — INVALID SDSLIB COMMAND ID
UNKNOWN ERROR PASSED, CODE = XXXX

# Appendix B

# Completion Messages

The following messages are issued by the assembler upon completion of processing. In addition to these messages, a number of messages are issued to the user in the assembly listing file and/or in the file specified in response to the ERROR ACCESS NAME prompt of the XMA procedure.

For this set of messages, the internal message code and the message ID in this manual are identical.

I ASSEMBLR-0001 MACRO ASSEMBLY COMPLETE, ?1 ERROR(S), ?2 WARNING(S)

Explanation:
The macro assembler has completed normally, although there may have been errors or warnings generated from the source code.

User Action:
No action is required.

USH ASSEMBLR-0002 MACRO ASSEMBLY ABNORMAL TERMINATION

Explanation:
The macro assembler has terminated before completing the assembly of the source code. The exact nature of the error is explained by the message in the file specified in response to the ERROR ACCESS NAME prompt.

User Action:
The action to take depends on the message in the file specified in response to the ERROR ACCESS NAME prompt.

US ASSEMBLR-0003 MEMORY REQUIRED EXCEEDS SYSTEM CAPACITY

Explanation:
The macro assembler was unable to secure enough memory to complete the requested assembly. If there are any source lines in the file specified in response to the LISTING ACCESS NAME prompt, the assembler was unable to complete the cross reference.

User Action:
If the system memory is relatively small, it may help to run the assembly when the system is less busy. If there is no shortage of physical memory, reduce the memory requirements of the program. The major items that use memory are macros and symbols. If the program contains macros that have no parameters or other macro variables, consider replacing the macro calls by source lines which are brought into the progam using a COPY statement. If the program contains macros with large amounts of text, assemble the macros into a macro library and use the LIBIN statement or provide a library pathname for the MACRO LIBRARY

PATHNAME prompt. If the program has many symbols, break it into two or more parts (using the REF command for references between parts) and use the Link Editor to combine the parts.

USH ASSEMBLR-0004 END ACTION TAKEN BY MACRO ASSEMBLER

Explanation:
The macro assembler was forced to the end action address either by executing an instruction that caused a task error or by the user killing the task.

User Action:
If the end action was not forced by user action, call a customer representative for assistance.

USH ASSEMBLR-0005 ERROR ATTEMPTING TO OPEN THE SPECIFIED ERROR ACCESS NAME

Explanation:
The macro assembler was unable to open the file specified for the ERROR ACCESS NAME prompt.

User Action:
Check the response to ERROR ACCESS NAME to be sure the syntax is correct and that all directories in the pathname exist. If this does not correct the problem, call a customer representative for assistance.

USH ASSEMBLR-0006 ERROR ATTEMPTING TO ACCESS SYNONYMS

Explanation:
The macro assembler received an error from the SCI routine S$GTCA.

User Action:
This is an unexpected internal error. Call a customer representative for assistance.

# Appendix C

# Error Listing Messages

This appendix contains a list of error and warning messages produced by the assembler. Error messages are printed in the listing file, when an error is detected, with the statement where the error occurred. Warning error messages are written only to the error file and are not included in the listing. A dash is placed in column eleven of the listing where the warning error occurred. Warning messages do not include an indication of a previous warning or error. Note that a warning is a dash (-) in column II of the assembled program listing.

| Error Message | Possible Causes |
|---|---|
| Absolute value required. | |
| Attempt to index by register zero. | A warning |
| Bad access name syntax. | A warning |
| 'CEND' assumed. | A warning |
| Close ( ")" ) missing. | |
| Comma missing. | |
| Common table overflow. | Too many common segments used (127 maximum). |
| Conditional assembly nesting error. | An if-then-else construct is in error. Conditions which could cause this are:<br>a. Missing ASMEND'S<br>b. Surplus ASMEND'S<br>c. Surplus ASMELS'S |
| 'DEND' assumed. | A warning. |
| Directory open error. | Check that any synonyms are valid and that no other processor is currently writing to the MARCO library. |
| Directory read error. | An I/O error was encountered while trying to read a macro library Directory. Verify that no other processor is currently writing to that macro library. |
| Directory required. | The access name specified is not an existing directory. Verify that all synonyms are correct and that the macro library does indeed exist; it can not be auto-created. |

| Error Message | Possible Causes |
|---|---|
| Directory write error. | Verify that no other processor is currently writing to that macro library. |
| Displacement too big. | An instruction requiring an operand with a fixed upper limit was encountered which overflowed this limit. An example is the 'JMP' instruction, whose single operand must evaluate to within >7F words distance from the current program counter. |
| 'DSEG' assumed. | This is a warning that the following two statements have the same result:<br>CSEG ' $DATA'<br>DSEG |
| Duplicate definition. | a. The symbol appears more than once in the label field of the source.<br>b. The symbol appears as an operand of a REF statement as well as in the label field of the source.<br>c. An attempt was made to define a macro variable or macro language label which was previously defined in the macro. |
| Error expanding call. | The symbol in the operand field of the $CALL statement is not a defined macro. |
| Error on copy open. | The access name specified as the operand of copy directive can not be opened. Check that the synonyms are correct and that the file is not currently being written to by another processor. |
| Expression syntax error. | a. Unbalanced parentheses.<br>b. Invalid operations on relocatable symbols. |
| Indirect (*) missing. | |
| Invalid $ASG variable. | a. An attempt was made to change the length component of a variable.<br>b. An attempt was made to change the attribute component or the value component of a macro variable which was declared as a macro language label.<br>c. The target variable is not present or is not a symbol. |
| Invalid character in symbol - blank used. | A warning (Note 1). The legal characters to be used in symbols under SDSMAC are A-Z, 0-9, ";", and "$". |
| Invalid Condition | The List Search instructions require conditions to be specified as one of the operands. The following are legal conditions: EQ, NE, HE, L, GE, LT, LE, H, LTE, GT. |

| Error Message | Possible Cause |
|---|---|
| Invalid CRU or shift value. | A warning |
| Invalid directive in absolute code. | The directives PEND, DEND, CEND have no meaning in absolute code. |
| Invalid expression. | May indicate invalid use of a relocatable symbol in arithmetic. |
| Invalid macro expression. | Invalid construct in $ASG statement. |
| Invalid macro variable. | The target variable specified on a $ASG or $GOTO verb is not a valid target variable. |
| Invalid model statement. | A macro symbol in a model statement must be followed with either a colon operator (:) or end-of-record. |
| Invalid opcode. | The second field of the source record contained an entry that is not a defined instruction, directive, pseudo-op, DX-OP, DFOP, or macro name. |
| Invalid option. | A warning. The only legal options are:<br>XREF, SYMT, NOLIST, MUNLST, RXREF, TUNLST, BUNLST, DUNLST, FUNL<br>(or suitable abbreviation). |
| Invalid relocation type. | Only PSEG relocatable or absolute symbols are allowed as the operand of an 'END' statement. |
| Invalid use of conditional assembly. | A conditional assembly directive may not appear as a model statement. |
| Invalid use of REF'd symbol. | REF'd symbols may appear in expressions only under certain conditions (see the 990/10 manual). |
| Invalid $ASG expression. | The expression is not present. |
| Invalid $IF expression. | The expression either is not present or does not evaluate to an integer value. |
| Label required. | $NAME statements must begin with a label of maximum length 2. $MACRO statements must begin with a label of maximum length 6. |

| Error Message | Possible Causes |
|---|---|
| Macro definition discarded due to errors. | An error was detected during the assembly of the macro definition. Use of the macro name in succeeding lines will cause error messages. |
| Macro library read error. | A 'LIBIN' was in effect and the statement was a macro in a specified macro library, but an I/O error was encountered when reading it. |
| Macro library write error. | The current 'LIBOUT' library could not be used at completion of a macro definition. Check that the macro is not currently begin written by another processor. |
| Macro string overflow. | In building a concatenated string, the length of the string exceeded 225 characters. |
| Macro symbol truncated. | A warning. The maximum length for a macro symbol is two characters. The following are legal macro symbols: A, A.S, B2.SV.<br><br>The following are illegal macro symbols: CNT, CNT.A, PM2.SL. |
| Max macro nesting stack depth overflow. | a. A macro calls itself recursively more than the allowed maximum number of times.<br>b. More levels of macro calling have been used than the allowed maximum. |
| Memory exceeded. | The program counter overflowed the value >FFFF. |
| Missing $END. | |
| Model statement truncated. | A warning. When expanded, the model statement exceeded 80 characters in length. |
| Open '(' Missing | A parenthesized operand is required with the Extract Field, Extract Value, Insert Field, and Invert Order of Field Insert instruction. |

| Error Message | Possible Causes |
|---|---|
| Operand conflict PASS1/ PASS2. | During pass 1, the assembler defaults currently undefined symbols as register names if that symbol is used in an ambiguous way, as shown in the example below. If during the pass 2 it is discovered that the symbol was not a register name, this error results.<br>An example is:<br>BL SUB<br>. .<br>. .<br>. .<br>SUB EQU $<br>If this example had been coded as follows, no ambiguity would have existed due to the explicit "@" sign:<br>BL @SUB<br>. .<br>. .<br>. .<br>SUB EQU $ |
| Operand missing. | On instructions having a fixes number of operands, too few appeared before encountering a blank. On instructions having a variable number of operands, such as 'DATA', a comma may have been encountered with no operand following it. An expression extending beyond the 60th column could cause this problem. |
| 'PEND' assumed. | A warning. |
| Register required. | |
| String required. | |
| String truncated. | A warning. Check the syntax for the directive in question to determine the maximum length for the string. |
| Symbol truncated. | A warning. The maximum length for a symbol is six characters. |
| Symbol required. | |
| Symbol used in both REF and DEF. | This is a conflicting, duplicate definition. |
| Syntax error. | |
| 'TO' missing. | 'TO' is a required part of the syntax for the $ASG Macro verb. |

# USER'S RESPONSE SHEET

**Manual Title:** Model 990 Computer DNOS Assembly Language Programmer's Guide (2270508-9701)

**Manual Date:** 1 August 1981 **Date of This Letter:** _____

**User's Name:** _____ **Telephone:** _____

**Company:** _____ **Office/Department:** _____

**Street Address:** _____

**City/State/Zip Code:** _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

| Location in Manual | Comment/Suggestion |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

CUT ALONG LINE

**NO POSTAGE NECESSARY IF MAILED IN U.S.A.**
**FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL**

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD

Cover Part No. 2310002-0001

TEXAS
INSTRUMENTS